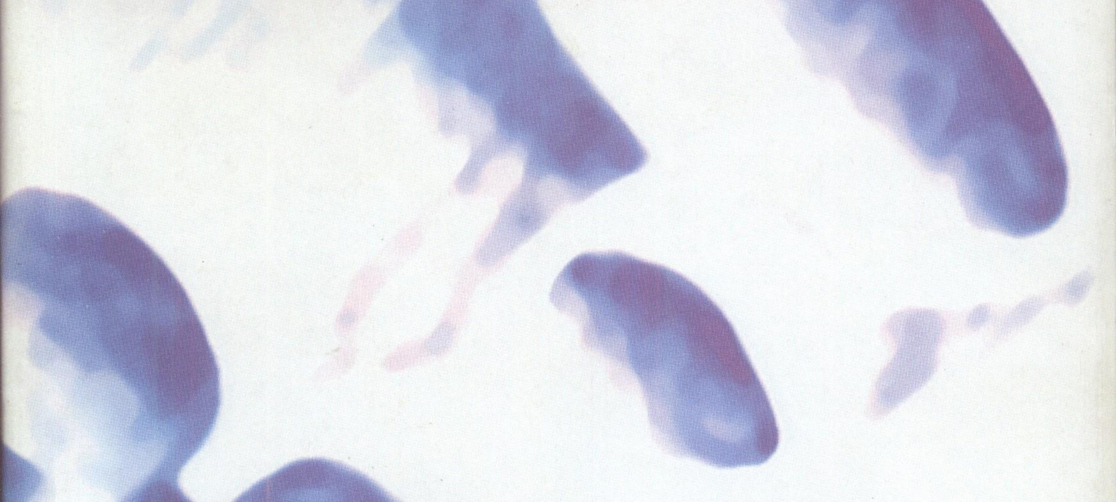
MS-DOSユーザーズ

386パソコン活用ブック

根津半蔵×星野進二◉共著



386/486パソコン上で新しいMS-DOS環境を!

i386/i486CPUのプロテクトモードを利用してMS-DOSを拡張するDOSエクステンダ

ソフトウェアの共存を実現するDPMI　マルチDOS環境のWindows 3

などi386CPUアーキテクチャからみたMS-DOS環境の充実をめざすための最新情報が満載!

ソフトバンク▶定価2,300円(本体2,233円)

MS-DOSユーザーズ

386パソコン活用ブック

根津半蔵×星野進二◉共著



SOFTBANK BOOKS SOFT BANK

MS-DOSユーザーズ

386パソコン活用ブック

根津半蔵×星野進二●共著

ソフトバンク

MS-DOSユーザーズ

386パソコン活用ブック

根津半蔵×星野進二●共著

ソフトバンク

まえがき

ほんの少し前までは，パソコンといえば処理能力が低く，もし高度な作業をするならばワークステーションを使うべき，という評価が一般的なものでした。しかし，パソコンの心臓部であるインテル社の 86 系 CPU が，i386/i486 と進歩したおかげで，いまではパソコンもワークステーション並みの処理能力を得られるようになりました。それでは，能力がワークステーション並みになった 386/486 パソコンは，現在，どのように活用されているのでしょうか？

MS-DOS の環境から i386/i486 独自の機能を利用する環境へと移行している過渡期のいま，さまざまな情報が氾濫しています。ユーザの中には，すぐに飛びつかず，じっと静観している人もいるのではないでしょうか。それでも多くのユーザが 386 パソコンを選択するのは，高速な 8086 パソコンとしても使えるからです。i386/i486CPU は，先進的なコンピュータ環境をサポートするための機能を数多くもち，「パソコン＝貧弱な MS-DOS 環境」という図式を打ち破れる(8086CPU とは比べものにならないほど)高機能なものです。確かに，80286 を CPU としてもつパソコンが登場したときは，それに対応する MS-DOS 上のソフトウェアがほとんどなかったために，多くは高速な 8086 パソコンとして使われていました。しかし，これも 386/486 パソコンとなれば，事情が異なってきます。なぜなら，MS-DOS をマルチにしたり，MS-DOS 上で 1M バイト以上のメモリを簡単に扱えたり，または MS-DOS を仮想記憶化したりなど，i386CPU 固有の機能を積極的に利用しながら MS-DOS を拡張するソフトウェアが，数多くあるからです。

そこで本書では，そうした背景をふまえながら，MS-DOS 環境における i386CPU の意義をテーマに，解説を進めていきます。とくに，

- **i386CPU は，8086CPU や 80286CPU とどう違うのか**
- **i386 を CPU としてもつと，MS-DOS 環境はどう進化するのか**

という 2 点に重点をおいています。8086 や 80286 と比較することによって i386 アーキテクチャの特徴を捉え，それを基本にしながら，現在利用可能な i386 対応ソフトウェアの仕組みを理解することを目的としています。さらには，これから次々と登場する「MS-DOS や Windows 関係の i386 対応ソフトウェアが，先進的な CPU の機能をどう活用しているのか」を理解する力をつけることも，ねらいのひとつです。

たいへん高機能な CPU である i386 のすべてを説明し尽くそうとすれば，膨

大な量の文章を必要とします。命令セットを含めてリファレンス的に機能を拾い上げている類書も，すでにいくつかあります。そのような命令セットなどの辞書的役目はほかに譲って，本書では解説項目を，MS-DOS 環境下に厳選して構成しました。現在，そして今後登場する i386/i486 対応とうたったソフトウェアにおいて，「機能上，なぜそのようなことが可能なのか」ということを理解するために必要な知識に限定し，その知識と疑似経験を体系づけながら身につけられるよう配慮しました。

本書を読むに当たっては，まず 8086 の概要は理解していることを前提にしています。たとえば，DS のようなセグメントレジスタ/AX といった汎用レジスタ/INT 命令でソフトウェア割り込みをかける，といった程度の知識は理解しているものとします。また，MS-DOS の基本的な知識や，ファンクションコールを INT 命令で行うことなども，予備知識として必要です。総じていえば，8086 パソコン上でプログラムがどのように動作するかを知っているものとしています。

主な構成としては，第 1 章で i386 の特徴と MS-DOS 環境下での位置づけを行い，第 2 章では，前章で述べた i386 の特徴的な機能をより詳細に説明します。このふたつの章で，i386 アーキテクチャを理解できるでしょう。第 3 章では，i386 上での MS-DOS 拡張を説明します。プロテクトモードの DOS アプリケーションを実現する DOS エクステンダや，それらがマルチ仮想マシン上で共存を実現するための規約である DPMI などが，話の中心になっています。第 4 章では，Windows 機能の変遷を通して，Windows 3.0 と i386 とのかかわりについてみていきます。後半の章で，i386 アーキテクチャを利用した MS-DOS の 386 向けプログラムが，どのように動作するのかを理解できるでしょう。なお，本書における例は，i386 搭載の PC-9801 と，その対応ソフトウェアを使用しています。

みなさんがこの本を読まれて，i386/i486 によって飛躍する，これからの新しい MS-DOS 環境への理解を深めていただければ幸いです。

最後になりましたが，この本を執筆するに当たって協力していただいたテックソフト&サービス社の菅原氏，ソフトバンク書籍編集部の棚木氏など多くの方々に感謝します。

1992 年 4 月

著者

CONTENTS

第1章

概説

第2章

386の主な機能

第3章

MS-DOSと386

第1章

概説

1 386/486の将来性

1.1 i386CPU登場に至るまでのCPU環境の紆余曲折

ほんの数年前までは珍しかった386パソコンも，いまでは新製品のほとんどを占めるようになりました。デスクトップタイプはもちろん，ノートブック型パソコンでさえ，最初に登場した386パソコンの性能を超えるものが現れています。さらに，i386CPU(以下，386と記す)の上位互換で，より優れた性能を発揮するi486CPU(以下，486と記す)を搭載したパソコンも，すでに高根の花ではない価格で揃ってきています。このようにCPUの性能が向上する中，残された問題は，その利用方法でした(fig.1.1)。

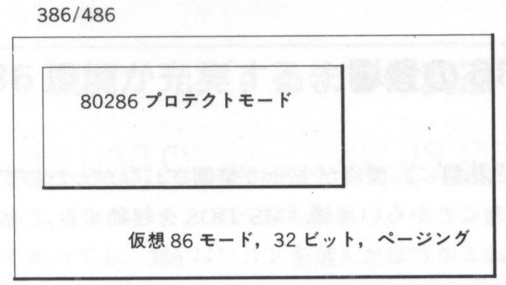
回 OS/2の問題点

80286が登場したとき，それに対応する新しいオペレーティングシステム(以下OSと略す)としてOS/2が登場しました。しかし，それは新時代の幕開けというわけにはいきませんでした。

OS/2はMS-DOSに比べると，はるかに本格的なマルチタスクOSでした。MS-DOSもOS/2の環境下で動作させることが可能です。ところが，80286は，CPUとしての機能が十分でなかったために，OS/2上でMS-DOSを起動すると，OS/2は止まってしまいました。たとえOS/2が先進的なOSであるとはいえ，止められてしまっては出る幕がありません。MS-DOSが同居することで，システムの信頼性が著しく下がってしまうようでは，安全で高機能なOSとしてOS/2を採用したい人にとっては，MS-DOSをOS/2下で動かすことを避けなければなりません。

また，逆にMS-DOSをバックグラウンドにしてOS/2に制御を移すと，今度はMS-DOSが止まってしまいます。OS/2下のMS-DOS環境は，かなり独

● fig.1.1 ● 80286/386/486の関係



立性が高く，マルチタスク環境の一部としては機能してくれなかったのです。もちろん，複数のMS-DOSを起動することもできません。Ver.2以前のOS/2では，OSが80286をターゲットとしているために，このような制限がありました。80286にしてみれば，OS/2の理想は，やや大きすぎた嫌いがあったわけです。

回 MS-Windowsの登場

そんな折しも，グラフィカルなユーザインターフェイス(GUI)が重視されはじめ，OS/2でGUIを実現する**プレゼンテーションマネージャ**が提供される頃には，MS-Windowsが登場してきました。こうなってくると，MS-DOSパソコンをとりまく環境は，混迷の色が濃くなりはじめてきます。

MS-Windows Ver.2はEMSメモリの使用を基盤としたMS-DOS用のGUI環境で，MS-DOSの延長線上につくられました。それは，GUI環境ではかなり先をいっていたMacintoshとよく似た環境を提供するものとして注目されました。

また，MS-Windows Ver.2からは，MS-Windowsを終了しなくても利用できる**DOS環境**というものがありました。しかし，これはOS/2から起動する

MS-DOSよりも，事情は深刻でした。つまり，MS-DOS用のメモリがあまりない環境であったため，できることがきわめて制限されてしまい，DOS環境とは名ばかりのものだったのです。

1.2 386の登場

なぜ，80286を基盤に，環境が十分に整備されなかったのでしょうか？ いままで述べてきたことからいえば,「MS-DOSを起動する」ことがキーワードであるといえます。

80286の機能では，どんなOS，GUI環境を用意しても，そこから起動できるDOS環境は何の保護も受けない80286**リアルモード**のMS-DOSです。しかも，80286ではリアルモードから**プロテクトモード**への移行は命令で可能ですが，逆にプロテクトモードからリアルモードに移行するための都合のいい命令はなく，CPUリセットをしなければなりません。

保護された安全なモードから何の保護もないモードへ，簡単に移行できるほうがおかしいのですから，CPUの設計としてはこれでもいいのですが，プロテクトモードの**マルチタスク環境**とMS-DOSを共存させたいという需要が厳然としてある以上，これでは問題です。すべてのアプリケーションをOS/2用に買い直すことなどはたいへんですから，このような需要があるのは当然といえます。しかし，80286のアーキテクチャでは，プロテクトモードとMS-DOSを同時に共存させることには対応しきれないのです。

この点386は，プロテクトモードとMS-DOSの混在という点を見直したCPUといえます。これには，80286と同じリアルモード，プロテクトモードのほかに，**仮想86モード**と呼ばれるモードがあります。仮想86モードというのは，簡単にいえば80286で採用されたマルチタスクの1タスクであり，その1モードです。これは，あるひとつのタスクで8086マシンを仮想的に実現する機能で，386用のマルチタスクOSが，プロテクトモードのままで，MS-DOSをひとつのタスクとして起動できることを狙いとしています。現在そうしたものの代表選手として，MS-Windows Ver.3.0とOS/2 Ver.2があります。ともに80286時代の反省をバネに，386用につくり直されたものです。

386は80286よりも高速なものが提供されているので，システムソフトウェアとしても高機能で実用的なシステムが構築できるようになっています。486ならば，386の性能をはるかに上回り，文句なしでしょう。

1.3 386環境が充実するまでの変遷

386パソコンが出たばかりの頃の使い道といえば，EMSエミュレータとMS-Windows/386くらいでした。それでも新規にパソコンを購入する際は，少しくらい高価であっても，386パソコンを選んだほうがよいとされました。それは，将来性とCPUの性能による即効性，さらにEMSがメモリボードなしに実現されることから，80286パソコンとEMSメモリボードをセットで購入するよりも386パソコンを選んだほうがよい，と判断されたからです。いまでは，黙っていても386パソコンです。

それでも現在のパソコン環境を見渡せば，本来の機能を生かした386パソコンというよりは,「高速な8086パソコン」として使用されている場合のほうが多いのではないでしょうか。すでに述べたように，386は優れたプロテクトモードをもつOSにおいて，MS-DOSの混在を許す機能をもちます。しかし，そうした環境が登場するまでには，実にさまざまなことが試みられてきました。

まず，ひとつには**EMSエミュレータ**です。これはMS-DOSを386の仮想86モードで実行させ，プロテクトメモリをEMSメモリとして利用するものです。これには，386の**ページング機能**が大きな役割を果たしています。386パソコンを手にした人が，CPUを386として最初に使用するには手ごろなプログラムでした。同様に，かなり立派なプログラムとして**DOSエクステンダ**と**386プロテクトモード用コンパイラ**があります。DOSエクステンダとは，MS-DOSから386プロテクトモード用プログラムを実行するための環境プログラムの一種です。

ほかにも，MS-Windows Ver.2上のMS-DOSを，仮想86モードで実現したMS-Windows/386があります。これは8086用のWindowsから比べれば，起動されるひとつひとつのMS-DOSが高い独立性をもち，メモリも多く取ることができるようになっています。

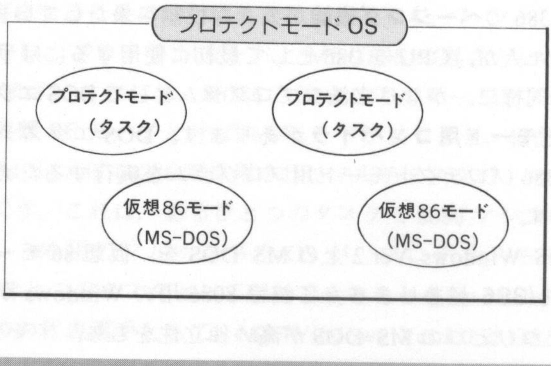
しかし，こうしたアプローチは，386 パソコンユーザに対して，ばらばらであったといってもよいでしょう。それぞれのニーズには応えていましたが，それらをまとめた統合環境というレベルにまでは至りませんでした。その結果，それぞれのユーザが増えるうちに，同時使用による不都合が生じはじめたので，混在のための規格が決められました。

いま，もっとも注目される規格はDPMI（DOS Protected Mode Interface）です。これは仮想 MS-DOS を提供するあらゆる 386/486 用プロテクトモード環境に適用されることが望まれます。486 は，386 と 100% のバイナリ互換です。そのため，386 用のソフトウェアが同じクロックなら 3 倍近くの性能を期待できます。DPMI にそって 386/486 のプロテクトモードを利用するアプリケーションが開発されれば，386/486 パソコンのリアルモードでの MS-DOS と同様に，仮想 MS-DOS でも問題なく実行されることが保証されるわけです。

このような使われ方が，現在の 386/486 パソコンを囲む状況ですが，これらをまとめると，386/486 パソコンがもつ将来性として，

- **プロテクトモード環境によるマルチタスク**
- **マルチ仮想 MS-DOS**

● fig.1.2 ● プロテクトモードとMS-DOSの混在



の2点が見えてくるでしょう。要は，先進的なCPUにふさわしい環境への移行と並行して，**連続的なMS-DOSのサポートの実現が必要とされる**ということです(fig.1.2)。

2 386/486の威力

この節では，386/486をCPUとしてもつパソコン(主にPC-9800シリーズ)で，8086や80286をCPUとしていたものと比べて，どのようなことができるようになったのか，CPUの機能にしぼって簡単に説明します。まず，大まかにまとめれば，次のようになるでしょう。

プログラミング上のメリット

【高速化】

- 32ビット処理ができる
- 64Kバイト以上のセグメントサイズを扱える

【保守性】

- 不正なポインタ指定を知ることができる(80286から継承)
- ハードウェアによりデバッグをサポートしている

システム環境のメリット

【高速化】

- 4Gバイトまでの物理メモリを扱える
- ページング機能を利用できる

【高度化】

- マルチタスクを提供できる(80286から継承)
- ユーザプログラムからシステムを保護できる(80286から継承)
- ほかのタスクからのタスクの破壊を防げる(80286から継承)

【MS-DOS】

- 仮想86を利用できる

2.1 プログラミング上のメリット

回 32 ビット処理

386 のレジスタは 32 ビットに拡張されていて，32 ビット演算ができるようになっています。通常，32 ビット演算を 16 ビットレジスタしかない CPU で行おうとすれば，32 ビットデータを半分に分け，それぞれについて演算し，さらに組み合わせる演算をする，というようになります。つまり，いくつかの命令を組み合わせて 32 ビット演算を行わなければならないのですが，386 ではそれらをひとつにまとめることが可能です。そのため，大きな数を高速に処理することが可能になりました。

たとえば，65,535 より大きな数同士の加減乗除も，32 ビットの演算命令を使用して 1 命令で求められるようになりました。

回 32 ビットデータバス

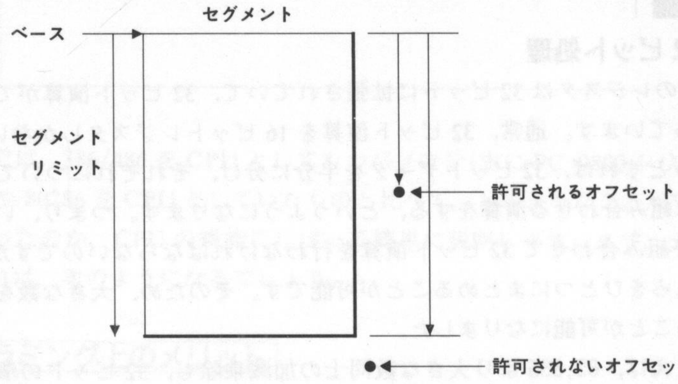
386 には 16 ビットデータバスのバージョン (386SX) もあるのですが，本書では本来の 32 ビットデータバスの 386DX について述べます。32 ビットレジスタとメモリの間でデータを読み書きするには，16 ビットのデータにして 2 回読み込むよりも，32 ビットで 1 回の読み書きをするほうが高速なのは明らかです。

現在，386 パソコン上では，MS-DOS が使用される場合が大半でしょうから，メモリのやりとりは 16 ビットになります。したがって，16 ビットデータバスの 386SX でも性能は変わりません。しかし，32 ビットデータを扱うアプリケーションを活用する場合には，1 回のメモリアクセスで触れるデータ幅が半分しかないので，それだけ遅くなります。

回不正なポインタの発見①(セグメントリミット)

プログラムのバグとしてよくあるのがポインタのまちがいです。関係のないところを指しているのに気づかずデータを書き換えてしまって，データ構造を破壊してしまうことがあります。そうした不正なポインタを，なるべく早く見つけることが，デバッグの際の重要なポイントです。

● fig.1.3 ●セグメントリミット



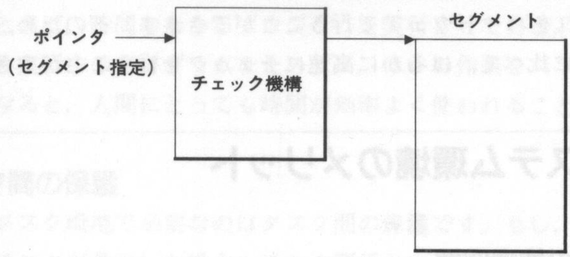
これに対して386では，80286から継承された**セグメントリミット**がそれを防ぐ役目の一部を果たします。セグメントリミットはセグメントそれぞれに与えられ，セグメントへのアクセスの際のオフセットを制限します。これが不正なオフセットによるアクセスを防ぎます(fig.1.3)。

8086ではセグメントのサイズは64Kバイトに決められていますが，386ではセグメントの大きさを指定することができます。そのため，実際に利用しているサイズを指定して，その範囲を超えたアクセスを拒否することができます。たとえば，1Kバイトのデータセグメントをデータ領域として割り当てた場合，その大きさを超える2000というオフセットは8086では拒否されません。その結果，関係のない場所にアクセスしてしまいますが，386ではセグメントリミットを1023としておけば(1Kバイトのセグメントの有効オフセットは0から1023)，このアクセスはエラーではねられます。

回不正なポインタの発見②(タイプ)

不正なポインタが，現在の命令ではアクセス不能なセグメントへのポインタである場合もあります。たとえば，データセグメントのつもりでコードセグメントへのポインタを設定する場合などです。

● fig.1.4 ●ポインタのタイプチェック



セグメントのタイプは，プログラムが自分のコード領域を書き換えてしまうようなことなどを見つけることができます（80286から継承した機能）。そのチェックは，セグメントに実行可能セグメントかデータセグメントかの2種類を設定することなどで行います。

8086では「アドレスAから始まる大きさBのメモリ領域をコードセグメントとして使う」という，プログラムにとっての暗黙の規約があるだけでした。あくまでも暗黙の規約であるため，プログラム自身が知らずにそのセグメントをデータセグメントとして使っても，それは知らずに行ったことですから使えてしまいます。386では，このセグメントの使い分けが暗黙の規約ではなく，明示的に指定します。そのため誤ってコードセグメントをデータセグメントとして指定し，書き込みを行おうとすればエラーになり，バグの箇所を早く見つけることができます（fig.1.4）。

回デバッグのサポート（デバッグレジスタ）

データのある領域がいつのまにか変わっていて，データ構造が壊れるという現象が，バグのあるプログラムにはよく見られます。そのデータ破壊の影響が顕在化するのは，破壊のずっとあとの場合が多いので，不正なアクセスを検出しても，その原因となったコードの実行からはあまりにも時間が経ちすぎていて，原因を発見するのに非常に手間がかかります。

そんなとき，破壊されたデータに書き込みを行っているすべてのコードの実

行を発見して，その場所で止めたいと思うことがあります。しかし，ソフトウェアによってそれを実行しようとすると，とてつもなく遅くなり，よほど調べるコード領域を絞り込まないかぎり，実用には耐えません。

386では，これをハードウェアで行うことができます。そのため，ソフトウェアで行う場合に比べて，はるかに高速にチェックを行うことができます。

2.2 システム環境のメリット

◻4Gバイトの物理空間

386では32ビットの物理アドレス空間をもちます。これは，4Gバイトもの広大な空間です。8086では1Mバイト，80286では16Mバイトであったのを考えると，いかに大きな物理空間に拡張されたかがわかります。

最近，CPUのパワーが向上するに従って，プログラムはますます複雑な作業を処理することが可能になり，結果としてプログラムも複雑になり，扱うデータの量も増えてきています。そこで，386では時代の要求に応え，大量の物理メモリをサポートできるようになりました。

パソコンでは，まだGバイトはおろか，数百Mバイト，数十Mバイトも物理メモリが搭載されることはあまりありませんが，ワークステーションクラスなら数十Mバイトのメモリは珍しいことではありません。これはマルチタスクを効率よくサポートするために，多くの物理メモリを必要とすることからきています。逆にみれば，多くのタスクが，実用に耐える速度で実行されるくらいCPUが高速になれば，多くのメモリを必要とするといえます。

◻マルチタスク

マルチタスクは同時に複数のプログラムを実行させる環境で，CPUが効率よく使用されます。80286に導入された機能ですが，386でも継承しています。

まず，マルチタスクになることのメリットは，CPUの利用効率が向上することです。あるプログラムが何かを待たなければならなくなったときに，マルチタスクの威力が発揮されます。マルチタスクでない環境では，その待ちが終了するまでの間，CPUはほかに何も行いませんが，マルチタスク環境下では

CPUがほかの実行可能なプログラムを実行できるので，CPUはムダになりません。

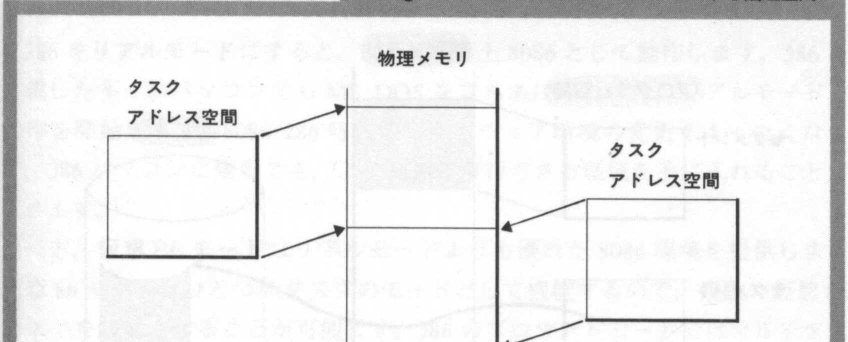
また，別な面として，パソコンユーザの待ち時間が減る，ということがあります。たとえば，時間のかかるプログラムを実行させておき，その終了を待つ間，それら実行中のプログラムを止めずに，ほかの作業をパソコン上で行います。こうなると，人間にとっても時間が効率よく使われることになります。

回タスク間の保護

マルチタスク環境で必要なのはタスク間の**保護**です。もし，保護がなければ，ひとつのタスクが暴走した場合，ほかの関係ないタスクまでもすべて巻き込んでしまいます。これでは便利なマルチタスクといえども，使う気になれません。1タスクの暴走は，それ以上ほかのタスクに影響してほしくありません。

386では，マルチタスク機能と密接に関連するタスク保護を，80286から継承しています。このタスク保護では，タスクごとに別々なアドレス空間を設定することによって，タスクが別タスクの領域に触れられないようにしているのです。たとえば，あるタスクの8000番地への書き込みは，ほかのタスクの8000番地のデータへ基本的には何の影響も与えません。これはアドレス変換

● fig.1.5 ●別の物理空間にマッピングされる論理空間



機構によって，ふたつのタスクのそれぞれの8000番地が，別の物理アドレスにマッピングされるからです(fig.1.5)。

回セグメント単位でのシステム保護

タスクのシステム領域への自由なアクセスを許すことは，システム全体を危険に陥れます。アプリケーションが，システム領域のデータを書き換えてしまうかもしれないからです。

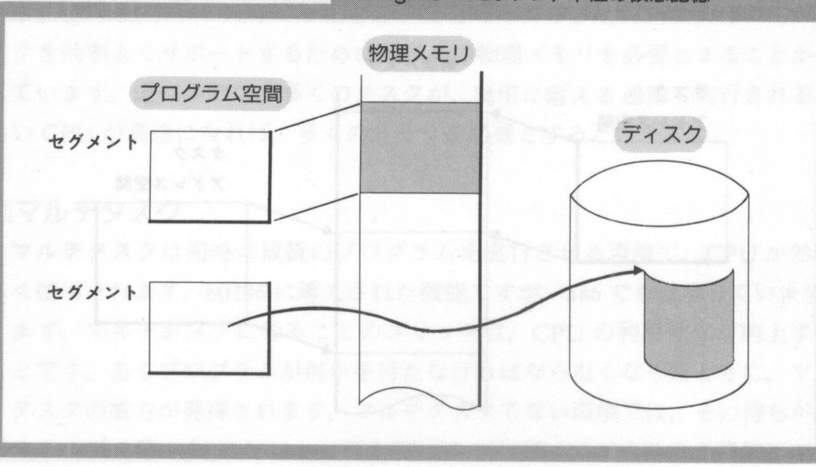
386では，80286から継承した**リング保護**を行います。リング保護はセグメントひとつひとつに**特権レベル**を設定して，アクセスを制限するものです。たとえばシステムレベルの特権を与えられたセグメントには，アプリケーションは自由に触れられないようにできます。こうしたリング保護のおかげで，タスクが暴走しても，システムまでおかしくなることはないのです。

回セグメント単位での仮想記憶

386ではセグメント単位での**仮想記憶**が可能になっています(fig.1.6)。これも80286から継承した機能です。

仮想記憶環境では，実際に搭載されている物理メモリ以上のメモリ空間を利

● fig.1.6 ●セグメント単位の仮想記憶



用できるため，少ない物理メモリでより大きな，より多くのプログラムを実行することが可能です。仮想記憶をサポートするためには，アクセスされるメモリ内容が実際に物理メモリにあるのか，あるいはディスク上にあるのかをチェックする機構が必要です。

80286から継承された機能では，セグメント単位でこのチェックを行います。つまり，アクセスされるセグメントが物理メモリ上にあるかないかをチェックする機構があるのです。物理メモリ上にないとわかれば，ディスクから物理メモリ上の適当な位置に読み込みます。その際，物理メモリに空きがなければ，使われている頻度がもっとも低いセグメントから順に，物理メモリへ追い出して空き領域をつくります。

◻ MS-DOSのサポート

8086の上でもっとも多く走っているOSとしてMS-DOSがあります。386はMS-DOSよりも高性能なオペレーティングシステムが走ることを想定してつくられていますが，MS-DOSの豊富な資産を引き継ぐことも重視しています。この点に関して386では，具体的に次のふたつをサポートすることによって，MS-DOS資産を利用することを実現しています。

- **リアルモード(80286から継承)**
- **仮想86モード**

386を**リアルモード**にすると，386は事実上8086として動作します。386を搭載した多くのパソコンでもMS-DOSを立ちあげるときは，リアルモードで動作を開始します。8086/286時代のソフトウェア環境の変更をまったくなしに，386パソコンに変更でき，より高速に実行できる環境を手に入れることができます。

一方，**仮想86モード**はリアルモードよりも優れた8086環境を提供します。仮想86モードはひとつのタスクのモードとして機能するので，複数の仮想86タスクを起動させることが可能です。386のプロテクトモードではマルチタスクをサポートしますが，その1タスクとして8086の環境をつくることができるのです。また，仮想86モードでは386の一部の機能がそのまま有効になっています。たとえば，ページング機能を利用して**仮想**EMSを実現し，メモリ

ボードを購入しなくても，386パソコンだけでEMSメモリが利用できます。MS-Windows Ver.3.0のエンハンストモードのように，386で動作するマルチタスク環境では，タスクを仮想86モードで起動することによって，Windowsのアプリケーションと MS-DOS アプリケーションとを，メモリを十分に確保しながら混在して利用できるばかりか，複数のMS-DOSを同時に起動することまでも実現しています。

回ページング機能

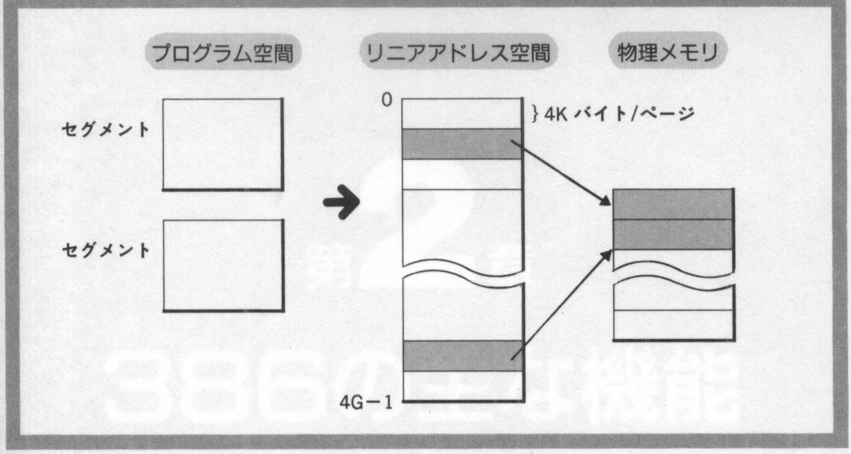
386で新しく付け加わったのが**ページング機能**です。

ページングはワークステーションなどに搭載されるプロセッサでポピュラーな機能です。この機能のおかげで，よりきめの細かいメモリ管理が行えます。まず，第一に386では，このページングを有効にしたり，無効にしたりできます。ページング機構が有効な環境では，プログラムからみると，自分のアドレス空間＝物理アドレス空間ではありません。また，ページングではセグメンテーションで生成されたアドレス空間と物理空間をさらに切り離します。具体的にいえば，セグメンテーションで生成されたアドレス(リニアアドレス)を，さらに物理アドレスに変換する機構がページングです(fig.1.7)。

386のページングには大きく分けてふたつの機能があります。ひとつはページ単位の仮想記憶のための機能で，もうひとつはページ単位のシステム保護です。ページングでは，メモリ空間を**ページ**と呼ばれる4Kバイトの領域に区切ります。これをページ単位の仮想記憶では，ページ単位にメモリとディスクとの間のやりとりを行います。これは，ページ単位でメモリ上にあるかどうかを記録できることで可能になっています。一方，システム保護については，スーパバイザとユーザの2種類を設定して，アプリケーション(ユーザ)がシステム(スーパバイザ)のページにアクセスできないようにできます。386では，セグメンテーションと合わせると，2段階のメモリ管理が行われることになります。

ページングを利用している身近な例としては，EMS **エミュレータ**があります。EMSエミュレータでは，EMSページが16Kバイトであることから，386の物理ページ4枚分を，EMSページ1枚にみなして管理します。

● fig.1.7 ●リニアアドレス空間のページ化



このように386とは，80286で取り入れられた先進の機能を継承しながら拡張したものであり，さらに独自の先進的なアーキテクチャをもったCPUであるといえます。

EMSメモリが利用できます。

スク環境…で，Windows アプリケーションとMS-DOSアプリケーションとを，メモリを十分に…
…複数のMS-DOSを同時に…
…でも実現しています。

ページング

386で新たに付け加わったのがページング機能です。
ページングプロセッサ…などに搭載されるプロセッサで…
…機能です。この機能のおかげで…細かいメモリ管理が…
す。特に386では，このページング機能により，物理…

ス空間＝物理アドレス空間ではありません。また，ページングではセグメ…
…に物理アドレスに変換する機構がページングです(Fig 1-11)。

386のページングには大きく分けてふたつの機能があります。ひとつはページ単位の仮想記憶のための機能で，もうひとつはページ単位のメモリ保護です。ページングでは，メモリ空間をページと呼ばれる4Kバイトの領域に分割します。これをページ単位の仮想記憶では，ページ単位にメモリとディスクとの間のやりとりを行います。これは，ページ単位でメモリ上にあるかどうかを記録することで可能になっています。一方，システム保護については，スーパバイザとユーザの2種類を設定して，アプリケーション（ユーザ）からスーパバイザのページにアクセスできないようにできます。386ではセグメンテーションと合わせると，2段階のメモリ保護が行われることになります。

ページングを利用している身近な例としては，EMSエミュレータがあります。EMSエミュレータでは，EMSページが16Kバイトであることから，386の物理ページ4枚分を，EMSページ1枚にあたるように管理します。

第2章

386の主な機能

1 セグメンテーション

386のプロテクトモードは80286と同様，8086に比べてセグメント管理面でも強力なものになっています。中でも特徴的なのは,

- **グローバル空間とローカル空間の分離**
- **各種セグメントの保護**

です。

1.1 セグメントディスクリプタによるセグメント指定

8086でセグメントを指定するためには，16ビットのセグメントレジスタでセグメントベースを指定しました。セグメントレジスタの値は，パラグラフと呼ばれる物理空間を16バイトごとに区切った単位によって指定されます。セグメントのオフセットも16ビットで指定されます。そのため8086では,

セグメントサイズ……64Kバイト
セグメントベース……64Kパラグラフ

となっていました。また，プログラムのアドレス指定と実際にアクセスされる物理アドレスの関係は,

セグメントレジスタ	**：オフセット**	**→**	**物理アドレス**
(16ビット)	**(16ビット)**		**(20ビット)**

でした。

それが80286からは，プロテクトモードでセグメントを指定するために，**セグメントディスクリプタ**を採用しています。このセグメントディスクリプタを

導入しても，プログラムからみたアドレス指定の方法は変わりません。

80286は，アドレス変換の際に，セグメントディスクリプタを有効に利用することで，

- **プログラムのアドレス空間と物理空間の分離**
- **セグメントの保護**

などを実現しています。それが，386のプロテクトモードのアドレス変換では，

セグメントレジスタ：オフセット
(13ビット)　　　　(32ビット)
→ セグメントディスクリプタ：オフセット
(32ビット)　　　　　　　(32ビット)
→ リニアアドレス
(32ビット)

のように，セグメントレジスタは直接セグメントベースを指さず，セグメントディスクリプタを指定します。セグメントディスクリプタはディスクリプタテーブルという表の1エントリで，セグメントレジスタの値は，その表中の1エントリを指定します。こうした仕組みについては，次項で詳しく説明します。

1.2 セレクタ

プロテクトモードでは，16ビットのセグメントレジスタの値は複数のフィールドをもち，次のようになっています。

15 … 3	2	1 0
INDEX	TI	RPL

INDEXとはディスクリプタテーブルのインデックスのことで，使用するディスクリプタを指定します。13ビットあるので，最大8K個指定できます。TI(Table Indicator)ビットは，2種類あるディスクリプタテーブルのうち，ど

ちらを使うか指定します。RPL(Requester's Privilege Level)は，リング保護のときに使います。

このようにプロテクトモードでは，セグメントレジスタの値はセグメントを指さずに**セグメントディスクリプタ**の選択のために使われるので，セグメントレジスタの値のことを**セレクタ**と呼びます。

1.3 ディスクリプタテーブル——GDTとLDT

回グローバル空間とローカル空間

プロテクトモードではアドレス空間に，

グローバル空間……主にシステム領域として使用
ローカル空間………主にタスク固有の空間として使用

の2種類があります。

マルチタスク環境では，アプリケーション間で壊しあうことを防がなければなりません。そのためには，通常，各タスクに固有のアドレス空間を割り当て，あるタスクのメモリへのアクセスが，ほかのタスクのメモリに影響を与えないようにします。このようにしておくと，あるタスクの500番地は，別のタスクの500番地とはまったく関係がなくなります。

また，システム空間は，全タスクから同じように見えるのが便利です。そのため，グローバル空間は通常，「システムにひとつ」となっています。

回GDTとLDT

セグメントディスクリプタの表であるディスクリプタテーブルは，グローバル空間とローカル空間に対応して，**グローバルディスクリプタテーブル**(GDT)と**ローカルディスクリプタテーブル**(LDT)の2種類があります。GDTはシステムにひとつ，LDTはタスクごとに用意されるのが一般的です。

GDTの場所はGDTレジスタ(GDTR)によって指定されます。GDTRには，GDTのベースアドレス(32ビット)と，GDTのサイズ(16ビット)が記述して

あります。

それに対して LDT の指定は，LDT レジスタ(LDTR：16 ビット)によって指定される GDT 中の LDT ディスクリプタによります。つまり，LDT はグローバル空間中に存在するセグメントのひとつです。LDTR(LDT レジスタ)によって指定された LDT には，セグメントサイズが記入してあり，LDT の大きさを 8K エントリよりも少なくできます。本章の「2　マルチタスク」(p.41)で説明するように，LDTR は各タスクに固有なタスクコンテキストの一部で，タスクは固有の LDT をもつことができます。

セレクタによって選択されるディスクリプタテーブルの各エントリは，GDT の場合はグローバル空間のセグメント，LDT の場合はローカル空間のセグメントのセグメントディスクリプタです。

1.4 セグメントディスクリプタ

セグメントディスクリプタ(fig.2.1)は，ディスクリプタテーブルのエントリで，8 バイトの連続フィールドです。BASE フィールド(32 ビット)，SEGMENT LIMIT フィールド(20 ビット)，DPL フィールド，P ビット，TYPE フィールド，さらに 80286 ではなかったビットなどが，決まったフォーマットで入っています。それぞれについて説明しましょう。

SEGMENT BASE

セグメントベースアドレス(32 ビットのリニアアドレス)です。

SEGMENT LIMIT

セグメントディスクリプタが関連するセグメント中で指定できる，オフセットの上限を指定します。20 ビットしかないため，バイト単位のリミットでは，最大 1M バイトのセグメントサイズしか指定できません。やはり，セグメントディスクリプタにあるビット(Granularity ビット)をセットすると，セグメントリミットの値を左に 12 ビットシフトし，0 から 11 ビットまでには 1 を入れ

● fig.2.1 ●セグメントディスクリプタ

データセグメント用ディスクリプタ

SEGMENT BASE 31..24	G	B	0	AVL	SEGMENT LIMIT 19..16	P	DPL	TYPE					SEGMENT BASE 23..16
								S = 1	E = 0	ED	W	A	
SEGMENT BASE 15..0						SEGMENT LIMIT 15..0							

コードセグメント用ディスクリプタ

SEGMENT BASE 31..24	G	D	0	AVL	SEGMENT LIMIT 19..16	P	DPL	TYPE					SEGMENT BASE 23..16
								S = 1	E = 1	C	R	A	
SEGMENT BASE 15..0						SEGMENT LIMIT 15..0							

システムセグメント用ディスクリプタ

SEGMENT BASE 31..24	G	●	0	AVL	SEGMENT LIMIT 19..16	P	DPL	TYPE					SEGMENT BASE 23..16
								S = 0	CODE				
SEGMENT BASE 15..0						SEGMENT LIMIT 15..0							

●無視されるビットを指す

た値をリミットとするように指定されます。最大4Gバイトのセグメントサイズを許します。

DPL(Descriptor Privilege Level)

ディスクリプタの特権レベルを表す2ビットです。アクセス権(AR)バイトの一部でもあります(詳しくは，本章の「3　保護」p.58を参照)。

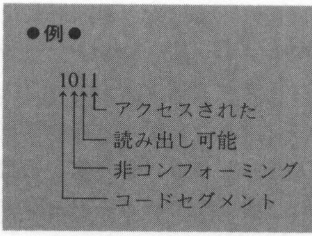
P(Segment Present)

セグメントディスクリプタの関連するセグメントが，物理メモリ上にあるかどうかを示す1ビットです。存在する場合は1に，存在しない場合は0にします。「P=0である」ときのディスクリプタは，DPL，TYPEの2フィールド(アクセス権バイトともいう)が有効なだけで，残りの部分(7バイト)は，すべてシステムが自由に使用できます。このセグメントのディスク内での位置などが，システムによって記されることになります。Pビットの管理はソフトウェア(OS)の責任です。「Pビットが0である」セグメントにアクセスすると，不在例外になります。このことを利用して，セグメント単位での仮想記憶が構築できます。システムセグメント(セグメントディスクリプタのSビットが0)でなければ，セグメントディスクリプタにはA(Accessed)ビットがあり，Pビットと協調して，セグメント単位の仮想記憶をサポートすることができます。

TYPEフィールド

これはアクセス権(AR)バイトの一部でもあります。Sビットが1の場合をまとめると，次のようになります。

TYPEのビット	各ビットの意味
1---	コードセグメント
1--	コンフォーミング
0--	非コンフォーミング
1-	読み出し可能
0-	読み出し禁止
0---	データセグメント
1--	下方伸長
0--	上方伸長
1-	書き込み可能
0-	書き込み禁止
---1	アクセスされた
---0	このビットがクリアされた



1　**A**（Accessed）……………（Sビットが1のときのみで，TYPEフィールドの一部）

セグメントがアクセスされたかどうかを示す1ビットです。このセグメントディスクリプタを指すセレクタがセグメントレジスタにロードされると，このビットが自動的に1になります。セグメントディスクリプタについて定期的に,「ビットが1かどうかを記録し，記録後0にする」ということを行っていれば，各セグメントの統計的なアクセス情報が取れます。メモリが足りなくなり，セグメントを追い出す必要が生じた場合，アクセス頻度の少ないセグメントから追い出すこと(LRU方式)ができます。

2　**S**（Segment Descriptor）

システム管理用のディスクリプタでないことを示す1ビットです。1の場合はアプリケーション用のセグメントディスクリプタであることを示し，0の場合はシステム用のディスクリプタであることを示します。

システム用のディスクリプタには大きく分けて，システムセグメントディスクリプタとゲートディスクリプタがあり，それぞれディスクリプタのフォーマットが異なります。また，システム用のディスクリプタであるかないかで，TYPEフィールドなどの解釈が異なります。

3　**E**（Executable）………………………（S=1のときのみ。TYPEフィールドの一部）

実行可能であることを示す1ビットです。1の場合に，セグメントディスクリプタの関連するセグメントが，コードセグメントであることを示します。0の場合は，データセグメントであることを示します。したがって，コードセグメントには書き込みを行えません。コードセグメントかデータセグメントかの違いによって，ほかのTYPEフィールドの解釈が異なります。

4-1　**R**（Readable）……………（S=1，E=1のときのみ。TYPEフィールドの一部）

コードセグメントが読み出し可能であることを示す1ビットです。1の場合に，セグメントディスクリプタの関連するコードセグメントが読み出し可能なことを示します。0の場合は読み出せません。読み出しを制限できるのはコードセグメントで，データセグメントはすべて読み出し可能です。

4-2 **W**（Writable）……………(S=1，E=0 のときのみ。TYPE フィールドの一部)

データセグメントが書き込み可能であることを示す1ビットです。1の場合に，セグメントディスクリプタの関連するデータセグメントが書き込み可能なことを示します。0の場合は書き込めません。書き込みを制限できるのはデータセグメントで，コードセグメントはすべて書き込み禁止です。

5-1 **C**（Conforming）…………(S=1，E=1 のときのみ。TYPE フィールドの一部)

コードセグメントがコンフォーミングセグメントであることを示す1ビットです。1の場合に，セグメントディスクリプタの関連するコードセグメントがコンフォーミングセグメントであることを示します。0の場合は通常のコードセグメントです(コンフォーミングセグメントについての詳細は，本章の「3 保護」p.58 を参照)。

5-2 **ED**（Expand-Down）….(S=1，E=0 のときのみ。TYPE フィールドの一部)

データセグメントが「下方伸長型」であることを示す1ビットです。1の場合に下方伸長のデータセグメントであることを示します。主に**スタック用**です。0の場合は通常のデータセグメントです。

スタックではデータをプッシュしていくと，スタックセグメント内オフセットは，リミット値から0の方向へ減っていきます。0を超えてプッシュしようとすれば，スタックフォールトが起きます。スタックフォールトが起きたあと，このスタックを伸ばして実行を続けることは不可能です。それは，SS を基準としているポインタがあるので，セグメントベースを移動させてセグメントの前方を拡張させることができないからです。かといって，リミットを変更してセグメントの後方を拡張しても，何にもなりません。

下方伸長データセグメントでは，セグメントのオフセット範囲はセグメントリミット＋1から0xffffffff となります。つまり，プッシュすると，スタックセグメント内オフセットは0xffffffff からリミット値＋1の方向へ減っていき，リミット値＋1を超えてプッシュしようとしてスタックフォールトが起きても，スタックを伸ばすためにはデータセグメントを拡張して，リミット値を減らすだけです。スタックポインタの値は引き続き使用できます。これを例で示しましょう。

いま，開始アドレスが0x10001000で，大きさ1024バイトの下方伸長データセグメントをつくったとします。下方伸長セグメントの場合，セグメントベースはセグメントの最後の次のバイトのアドレスを指し，この場合，0x10001400を指定します。スタックの開始地点はベース0x10001400からオフセット0xffffffffの位置ですから，32ビットの符号付き整数加算を行って，0x100013ffとなります(0xffffffffは −1です)。オフセットがリミット ＋1のときにセグメントの開始地点0x10001000となるように，リミットは−1025(0xfffffbff)とします。スタックを上方に1024バイト増やすには，現在のセグメントの前に1024バイト領域を確保して，リミットを−2049(=−1025−1024)にするだけです。

6　CODE………………………………………………………(S=0のとき)

ディスクリプタがシステム用である場合に，その種類を識別する4ビットです。次のようになっています。

システムセグメントディスクリプタを示すコード

(構成フィールド：ベース，リミット，G，P，DPL，AVL)

0x1 (0001)　アベイラブルな286 TSS
0x2 (0010)　LDT
0x3 (0011)　ビジーな286 TSS
0x9 (1001)　アベイラブルな386 TSS
0xB (1011)　ビジーな386 TSS

ゲートディスクリプタを示すコード

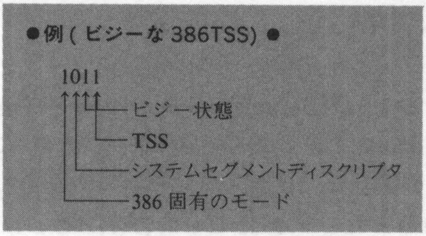
(構成フィールド：セレクタ，オフセット，ダブルワードカウント，P，DPL)

0x4 (0100)　286コールゲート
0x5 (0101)　タスクゲート
0x6 (0110)　286インタラプトゲート
0x7 (0111)　286トラップゲート
0xC (1100)　386コールゲート

0xE (1110) 386インタラプトゲート
0xF (1111) 386トラップゲート

0x0はCPUで使用されません。0x8 (1000), 0xA (1010), 0xD (1101)の3つのコードは，将来のCPUのために予約されています。これを無視すれば，4ビットを次のように見ることができるでしょう。

CODEのビット	各ビットの意味
1---	386固有のモード
0---	80286互換のモード
-1--	ゲートディスクリプタ
00	コールゲート
01	タスクゲート
10	インタラプトゲート
11	トラップゲート
-0--	システムセグメントディスクリプタ
-1	TSS
0-	アベイラブル
1-	ビジー状態
10	LDT



0xA (1010)と0xD (1101)が使用されないのは，80286/386でLDTディスクリプタとタスクゲートディスクリプタのフォーマットが同じためです。これは，それぞれのディスクリプタのもつ値がセレクタ値で，ともに16ビットであるからです。ベースアドレス，オフセット，リミットをもつディスクリプタの場合，80286と386とではビット幅が異なるので，4ビットの最初の1ビットで区別することになります。

TSS(タスクステートセグメント)ディスクリプタ，ゲートディスクリプタについての説明は，後ほど行います。

386拡張フィールド

1　G(Granularity)

セグメントリミット値を，ページ(4Kバイト)単位で解釈することを示す1ビットです。1のときは，セグメントディスクリプタの20ビットのリミット値が12ビット左シフトされ，下位12ビットには0xfffが挿入されます。したがって，リミットは0xfffから0xffffffffまで4Kバイトごとの指定になります。最小4Kバイト，最大で4Gバイトのセグメントをつくることができます。0のときは，バイト単位で解釈され，リミットは0x0から0xfffffまで1バイトごとの指定になります。最小0バイト，最大1Mバイトのセグメントをつくることができます。

下方伸長セグメントの場合は，セグメントサイズについて少し事情が異なります。G=0, B=0の場合(80286互換の下方伸長セグメント)，セグメントオフセットはリミット＋1から0xffffとなり，最小0バイト(リミット0xfffe)，最大64K−1バイト(リミット0)です。G=1, B=1の場合(386固有の下方伸長セグメント)，セグメントオフセットはリミットを左シフトして0xfffを挿入したもの＋1から0xffffffffとなり，最小4Kバイト(リミット0xffffe)，最大4G−4Kバイト(リミット0)です。

2-1　D(Default)……………………………………………………(E=1のときのみ)

386ネイティブモードのコードセグメントであることを示す1ビットです。1の場合は，セグメントディスクリプタの関連するコードセグメントが，386の命令セットと32ビットのオフセットであることを示します。0の場合は，80286互換の命令セットと16ビットのオフセットであることを示します。

2-2　B(Big)……………………………………………………(E=0のときのみ)

386ネイティブモードのデータセグメントであることを示す1ビットです。1の場合，セグメントディスクリプタの関連するデータセグメントが，64Kバイトより大きなセグメントを扱うことを示します。0の場合は，80286互換の64Kバイトを上限とするセグメントを扱うことを示します。

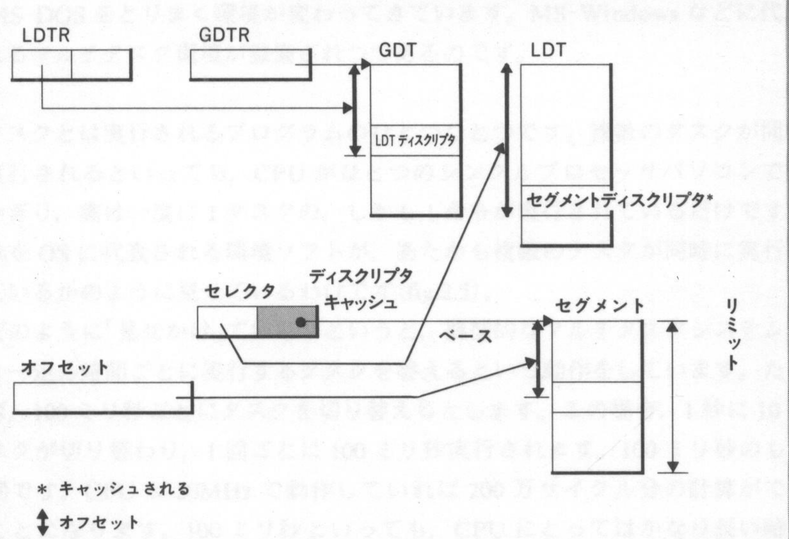
2-3　**AVL**(AVaiLable for Use by System Programmers)

システムが自由に使用してかまわない1ビットです。現在定義されていないフィールドでも使われる可能性がある場合は，勝手に使うと，将来の互換性が失われてしまいます。しかし，このビットは自由に使っても，将来のCPUでも使われる可能性がないことを保証したビットです。

1.5 ディスクリプタキャッシュ

プロテクトモードの「セレクタ：オフセット」の組からなるポインタは，ディスクリプタテーブルを引いて，セグメントディスクリプタをCPUに読み込みます。読み込む時点で，セグメントディスクリプタに対して各種チェックを行い，正しいセグメントディスクリプタであればベースアドレスとオフセットからリニアアドレスを生成します。

● fig.2.2 ●アドレス変換とディスクリプタキャッシュ



しかし，メモリアクセスごとにディスクリプタの読み込み，チェックを行っていたのでは，非常に遅くなってしまいます。それを回避するために，一度読み込んでチェックの終わったセグメントディスクリプタは，セレクタが変更されるまでディスクリプタキャッシュに保存されます。

このディスクリプタキャッシュは，各セグメントレジスタにひとつずつあります。ディスクリプタキャッシュのおかげで，同じセレクタが使用されるかぎり，メモリへのアクセスなしに「セレクタ：オフセット」からリニアアドレスへのアドレス変換ができます(fig.2.2)。

2 マルチタスク

2.1 マルチタスクの概念

マルチタスク機能という考え方はもうすでに定着しているかと思いますが，ここで，もう一度振り返って確認してみることにしましょう。マルチタスクは，簡単にいえば，

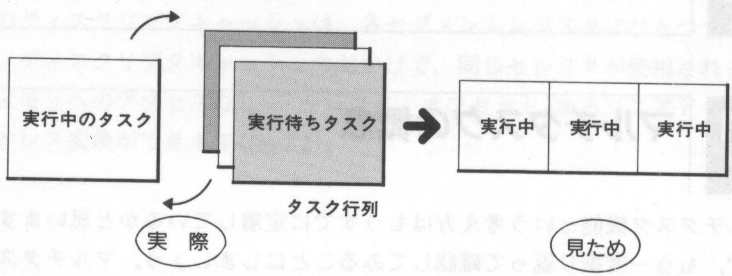
複数のタスクが同時に実行される

ということです。ワークステーションのOSとして有名なUNIXは，複数のプロセスが同時に実行されるマルチタスク(プロセス)OSです。それに対し，MS-DOSはシングルタスクのOSであることが知られています。しかし，最近MS-DOSをとりまく環境が変わってきています。MS-Windowsなどに代表されるマルチタスク環境が整備されつつあるのです。

タスクとは実行されるプログラムのひとつひとつです。複数のタスクが同時に実行されるといっても，CPUがひとつのシングルプロセッサパソコンであるかぎり，実は一度に1タスクの，しかも1命令が実行されているだけです。それをOSに代表される環境ソフトが，あたかも複数のタスクが同時に実行されているかのように見せているわけです(fig.2.3)。

どのように「見せかけ」ているかというと，典型的なマルチタスクシステムでは，一定の時間ごとに実行するタスクを替えるという動作をしています。たとえば，100ミリ秒ごとにタスクを切り替えるとします。この場合，1秒に10回タスクが切り替わり，1回ごとに100ミリ秒実行されます。100ミリ秒のもち時間です。CPUが20MHzで動作していれば200万サイクル分の計算ができることになります。100ミリ秒といっても，CPUにとってはかなり長い時間

● fig.2.3 ●見ためのマルチタスク



であることがわかるでしょう。たとえ 10 分の 1 の 10 ミリ秒ごとにタスクが切り替わったとしても，1 回の順番で 20 万サイクル分の計算ができます。

このように，マルチタスクでは，タスクが切り替えられて，

CPU による実行の機会が次々にタスク間で移り変わる

動作をしているのです。

さて，ここでふたつのキーワードを紹介しましょう。

あるタスクを実行途中で中断することを**プリエンプト**と呼びます。よく「タスクがプリエンプトされる」といった使い方をします。また，プリエンプトされたり，ほかの理由によって停止していたタスクが，再び実行の機会を得ることを**ディスパッチ**と呼びます。これは「タスクがディスパッチされる」といった使い方をします。

タスクは，存在すればすべてが CPU によって実行される対象となるとはかぎりません。タスクにはいろいろな状態があり，その状態によって実行されたりされなかったりします。タスクの状態でわかりやすいのは，**実行可能状態**または**待ち状態**です。

実行可能状態とは，その名の通り，いつでも走り出せる状態のことです。この状態のとき，タスクはCPUによって実行できます。実行可能なタスクが複数あると，それらは決められた順番で待ち行列をつくり，ディスパッチされます。これは，準備ができているタスクの行列ということで，レディキュー(readyqueue)とよく呼ばれます。この行列の作り方は，システムによってさまざまで，単純に割り当てられた時間分を計算したタスクが行列の後ろに回るか，タスクそれぞれに優先順位が与えられていて，全体の実行時間がその順位を反映するようにするか，などシステム設計の数だけあります。

こうした待ち行列をつくることを**スケジューリング**と呼んでいます。タスクを待ち行列に入れることを「タスクをスケジュールする」といいます。実行可能状態に対して，待ち状態にあるタスクはスケジュールされません。なぜなら，実行の機会を与えられても，実行できる状態ではないからです。

実行できない理由はさまざまです。ほかのタスクから送られる情報を待っていたり，ほかのタスクがもっている資源を解放してくれるのを待っていたり，注目している変数の値が条件を満たすのを待っていたり……。

基本的には何かを「待って」います。待ちの種類は，どれだけあげてもきりがありませんが，基本的な分類はあります。しかし，そうした分類については，OSの専門書に任せることにします。最近では，日本語版の参考書がかなりあるようなので，興味ある方は，ひもといてみられるとよいと思います。ここでは，「①タスクは何かを待っていて，②次には進めない」ことだけを理解していれば十分です。つまり，

タスクがスケジュールの対象となるかどうかは，その状態に依存

するということがいえるのです。

さて，これまでの説明で，典型的なマルチタスクでは複数のタスクがさまざまな状態遷移を経ながら，CPUによって間欠的に実行されることがわかっていただけたと思います。実行可能なタスクはそれぞれ，

①スケジュールされる(待ち行列に入れられる)
②たまにディスパッチされる(実行される)

③しばらくしてプリエンプトされる(実行が中断される)
④また①に戻る

これらを，タスクが実行可能でなくなるまで繰り返します。こうした間欠実行のメカニズムで動作していることは人間には知覚できません。たとえば，次のようなことを考えてみてください。

蛍光灯が高速に点滅しているようには見えない，あるいは実際には高速な紙芝居であるテレビの映像が連続な動きに見える，といったことを。

人間の感覚では，パソコンが1秒間に数十回以上もタスクを切り替えて，「順番に」実行していることには気づかないのです。昔のCPU，つまり，数MHzのクロックで動作し，1命令の実行にもずっと多くのサイクル数を要した時代には，逆にその切り替えが見えるほどでした。パソコンでのマルチタスクが実用的になったのも，最近のCPUの高速化によるものです。

おまけの話です。いくらCPUが高速になっても，速くなる分だけしたいことが大きく膨らんでいくので，高速なCPUが自分のタスクに回ってこないことに気づくことがあります。

先ほどの例で，実行可能なタスクが10個走り続ければ，タスクひとつに回ってくる実行の機会は，1秒に1回になります。1台を複数の人で同時に使う環境だと，たまにこうしたことが起きかねません。そのひとつがエディタのような対話的なプログラムであるような場合，1秒に1回しか実行の機会が回ってこないので非常に「重く」感じられ，それが逆にマルチタスクであることを感じさせます！　ここでは，10個のタスクの例を示しましたが，そうした状況は非常にまれです。普通の使い方では，同時に実行可能状態にあるタスクの数は，あまり多くはなく，たとえ数多くのタスクが同時に存在しても，それらが年がら年中実行されているということはほとんどないようです。たまにゴゴッと動いてはじっと眠る，というのがよくあるパターンです。

2.2 386によるマルチタスクサポート

回タスクのコンテキスト

80286から，マルチタスクを積極的にサポートするメカニズムがCPUに組み込まれています。386/486も，タスク管理については80286のメカニズムを踏襲しています。

386の話をする前に，もう少しタスク切り替えの仕組みについて説明しましょう。タスクにはタスクの**コンテキスト**というものがあります。コンテキスト＝文脈ですが，いい換えると，タスクが実行されるためにCPUが保持する情報といえます。保持する情報としては，主にタスクを実行しているときの，CPUのレジスタ内容をひとまとめにしたものです。

CPUにとって重要な情報の多くがレジスタに入っているため，

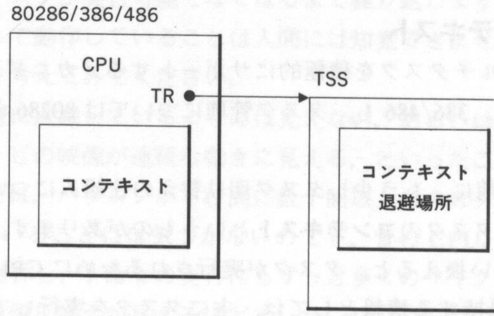
コンテキストはレジスタ内容

と考えてもよいと思います。

コンテキストは，マルチタスクにおいて重要な概念です。コンテキストとしてCPUの状態をそっくり保存することで，タスクの切り替えが可能になっているからです。たとえば，あるタスクが実行されているとします。そのタスクのもち時間がいっぱいになり，実行が中断(プリエンプト)されるときを想定します。その中断されるタスクが，次に実行する機会を得るとき，前回中断したところから実行を再開できるためには，必要な情報をすべてとっておかなければなりません。しかも，この場合タスクに自分が途中で中断されることを意識させてはなりません。それを回避するもっとも単純な方法は，タスクが中断されるときのすべてのレジスタ値をコンテキストとして保存することです。次回実行が再開するときに，その値を全部元のレジスタに戻しておけば，中断がなかったかのようにタスクの実行を再開できます。

386では，こうしたマルチタスクに必要なデータ構造のための，ハードウェアサポートを行っています。いままでコンテキストと呼んできた情報の集合体のために，**タスクステートメントセグメント**(TSS：Task Statement Segment)

● fig.2.4 ●コンテキストとTSSを指すCPU



というコンテキスト専用のセグメントを定義しています。TSSを指すセグメントディスクリプタを**TSSディスクリプタ**と呼びます(fig.2.4)。

TSSはシステムの資源であることから，グローバル空間内に置かれます。そのため，TSSディスクリプタはGDTのエントリになります。GDT内のTSSディスクリプタを指すためのレジスタが専用に用意されていて，これを**タスクレジスタ**(TR)といいます。つまり，現在実行中のタスクのコンテキストの場所は，

①タスクレジスタのセレクタ値がGDT内のTSSディスクリプタを指定
②TSSディスクリプタ中のベースがTSSのリニアアドレスを決定

され，データセグメントなどにアクセスする手順と同様の計算で求められます。386で定義されているTSSの主な内容は，次の通りです。

セグメントレジスタ群……………GS, FS, DS, SS, CS, ES
汎用レジスタ群　…………………EDI, ESI, EBP, ESP, EBX, EDX, ECX, EAX

フラグレジスタ……………………EFLAGS
命令ポインタ………………………EIP
バックリンク
ローカル空間………………………LDTR
ページディレクトリ………………CR3
特権0から2のスタック………SS2, ESP2, SS1, ESP1, SS0, ESP0
I/O許可ビットマップベース
トラップビット

はじめの4種類は，タスクが切り替わるごとに保存されるものです。これらはLDTRとともに，タスクがディスパッチされると，元のレジスタに復帰されます。バックリンクは，直前に実行されていたタスクのTSSのセレクタです。こうしたコンテキストの中にLDTRが入っていることを見ると,「タスク間保護を行うためにタスクが変わるとアドレス空間(ローカル空間)を変える」作業を，タスク切り替え時に自動的に行っていることがわかると思います。これ以外のTSSの領域は，システム依存の領域として使用されます。システム依存以外の，386で定義されている領域のサイズは104バイトあります。そのため386は，CPUがセグメントとしてTSSの最初の104バイトに確実にアクセスできるように，TSSディスクリプタのリミットが103以上であることをチェックします。このことからも，TSSはほかのセグメントと同様の扱われ方であることがわかるでしょう(fig.2.5)。

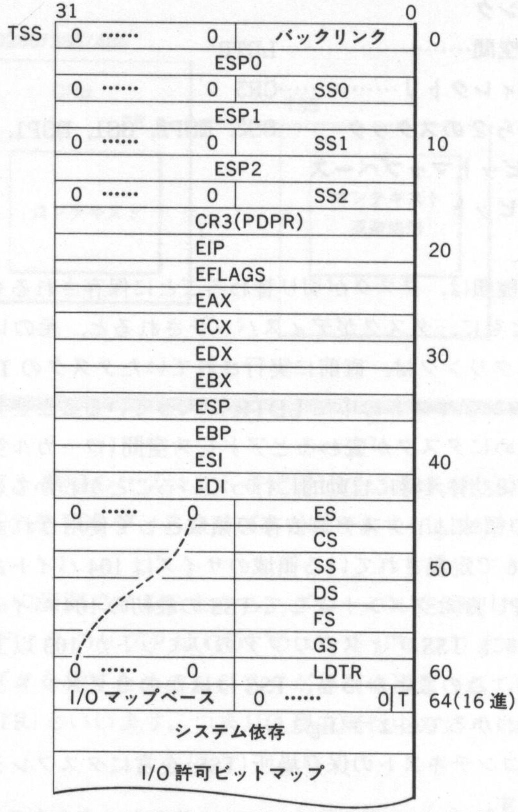
386では，コンテキストの保存場所(TSS)を常にタスクレジスタでもつことができるのです。

回タスクの切り替え

386は，ハードウェアで**タスク切り替え**をサポートしています。**ハードウェアサポート**とは，タスクを切り替える際に，自動的に現在のコンテキストを保存して，新たなコンテキストを復活させるという動作をCPUが行ってくれることです(fig.2.6)。

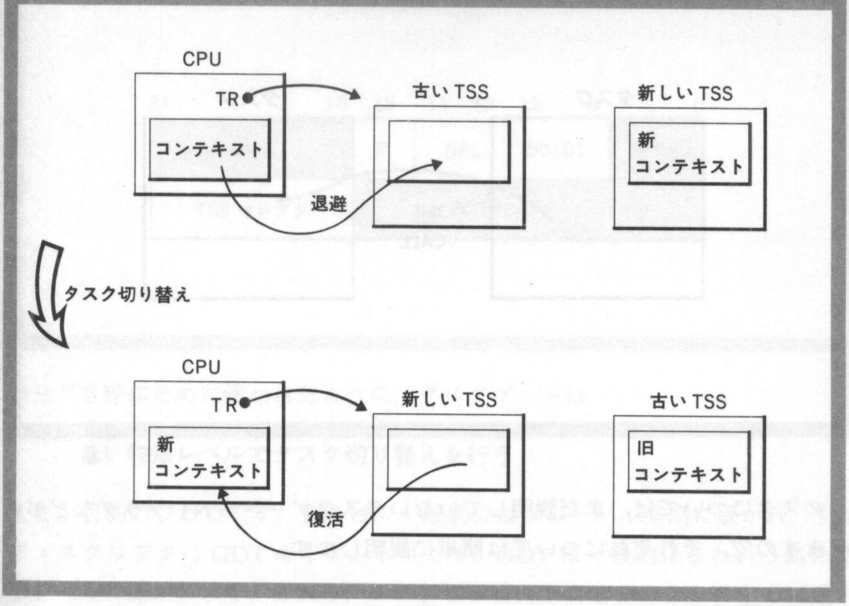
386で定義されているレジスタ群については，自動的に保存，復活が行われます。ハードウェアサポートがなければ，タスクのプリエンプト，ディスパッ

● fig.2.5 ● TSS の構造



チの責任をもつプログラムが，ソフトウェアでコンテキストの保存復活を行わなければなりません。ハードウェアサポート外として 386 のタスクコンテキストが保存してある TSS には，主に OS が使用する，システム依存のフィールドがあります。その部分については，何が書いてあるかを 386 は知らないので，管理はソフトウェア任せになります。以上をまとめると，タスク切り替えが起こると 386 は

● fig.2.6 ●自動的なコンテキスト退避/復活



自動的に現在のレジスタコンテキストを保存，新しいものを復活する

といえます。

それでは，タスク切り替えをどのように起こすのでしょうか。386 でタスクの切り替えが起きる場合には，次の 4 種類があります。

【JMP，CALL に関連】

(1) JMP か CALL の際に，セレクタが TSS ディスクリプタを指す

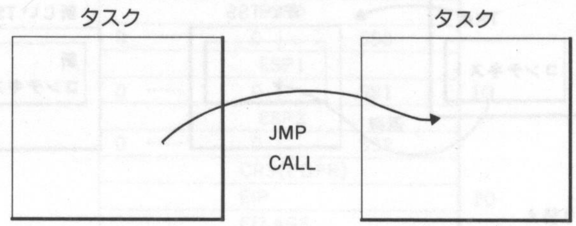
(2) JMP か CALL の際に，セレクタがタスクゲートを指す

【割り込みか例外に関連】

(3) 割り込みか，例外発生の際に，IDT 内の該当エントリがタスクゲートである

(4) IRET の際に，EFLAGS レジスタの NT フラグが 1

● fig.2.7 ● JMP, CALL



タスクについては，まだ説明していないタスクゲートやNTフラグなどがありますので，それぞれについては簡単に説明します。

JMP，CALLによるタスク切り替え

(1)は，セグメントを切り替えるJMP，CALLで飛び先のセグメントセレクタを指定(fig.2.7)する代わりに，**TSSセレクタ**を指定します。セレクタが指すディスクリプタに記載されているタイプで，TSSディスクリプタであることが判別できます。

(2)は，セグメントを切り替えるJMP，CALLで飛び先のセグメントセレクタを指定する代わりに，**タスクゲート**を指定します(fig.2.8)。セレクタが指すディスクリプタに記載されているタイプで，タスクゲートであることが判別できます。タスクゲートはゲートディスクリプタのフォーマットをもちますが，コールゲートのように飛び先のオフセットや引数の量などがなく，TSSセレクタが記載されています。タスクゲートを使ったJMP，CALLによるタスク切り替えは，このTSSセレクタが指すTSSディスクリプタでJMP，CALLを行うのと同じ効果です。では，なぜタスクゲートを一段入れるのでしょうか？

コールゲートが，低い特権レベルから高い特権レベルのコードセグメントの

● fig.2.8 ●タスクゲート

31　　16	15	14　13	12　　8	7　　0
未 使 用	P	DPL	00101	未使用
TSS セレクタ	未 使 用			

コードを呼ぶために使われたように，タスクゲートは

低い特権レベルでタスク切り替えを行う

ことが目的のひとつです。TSS は，一般的にはグローバル空間に置かれ，TSS ディスクリプタは GDT の 1 エントリです。GDT は一般的に 0 という高い特権レベルに置かれます。これは，タスク切り替えを含むシステムの中心に近い機能が，信頼性の低いソフトウェアから操作されないためです。TSS セレクタを使った JMP，CALL によるタスク切り替えも，低い特権レベルで行ってはなりません。しかし，タスクの種類によっては，低い特権レベルからでも切り替えを行いたい場合もあります。たとえば，アプリケーション自身で並列プログラミングを実現したいとき，並列化するルーチンひとつひとつをタスクで実現すると，ルーチン間で制御を移行させるときには，システムに頼らずにユーザプログラムレベルのタスクをほかのタスクへ切り替えたい場合が生じます。そのような場合に，切り替え先タスクの TSS を指すタスクゲートを使えば，特権 0 でなくてもタスクゲートにアクセス可能にしておくことで，ユーザタスクが自分で関連タスク間の切り替えを行うことができ，スケジューリングが可能になります。

CALL については JMP と異なる点があります。CALL では，また同じ場所に戻る可能性があるため，同じタスクの CALL の直後に戻ることが考えられます。そのために CALL されたタスクが，CALL したタスクを知っていなければなり

ません。そこでCALLでは，新しいTSSのバックリンクフィールドにCALLしたタスクのTSSセレクタを入れ，さらに新しいEFLAGSレジスタのNT(Nested Task)ビットを1にセットしておきます。NTビットを立てることによって，タスクに戻り先があることを示します(あとで説明する割り込み，例外でも同じ)。CALLしたタスクに戻る場合は，このあと説明するIRETを使います。

割り込み，例外によるタスク切り替え

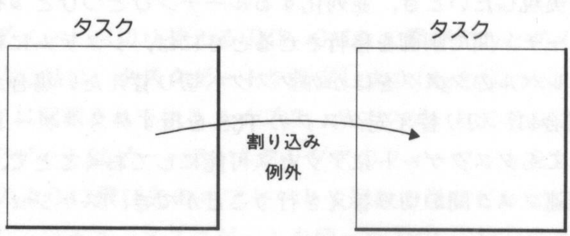
(3)もタスクゲートを使用します。この場合のタスクゲートの使われ方は，

割り込み，例外でタスク切り替えを起こす

ことです。386では割り込み，例外をタスクで処理することができます。また，割り込み，例外で特定のタスクを起動するといった使い方もできます(fig.2.9)。こうしたことを可能にする仕組みは，次の通りです。

386では割り込み，例外が生じると，IDT内の該当エントリ(ゲート)にそって割り込み，例外を処理するルーチンを呼び出します。その際，そのゲートにタスクゲートを置くことができます。そうすると，そのタスクゲートのIDの

● fig.2.9 ●割り込み/例外



割り込み，例外が生じたときに，タスクゲートに記載されたTSSセレクタをもとにタスク切り替えを起こします。ゲートを変更しないかぎり，同じIDの割り込み，例外に対して同じタスクに処理させたり，同じタスクを起動させることが可能です。この種類のタスク切り替えが起こるとき，CPUは，現在のTRの内容(TSSセレクタ)を，新しく起動されるタスクのTSSのバックリンクフィールドに保存し，さらに戻り先のタスクがあることを示すために，新しくロードされるEFLAGSレジスタのNTビットを1にします。

タスクのネストからの復帰

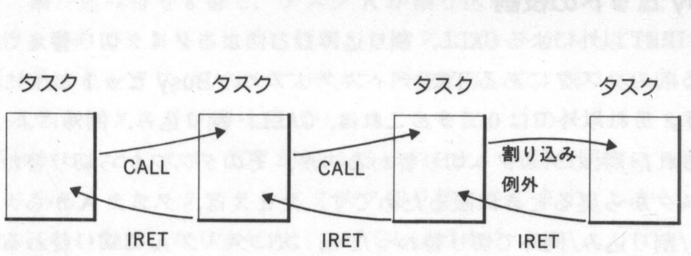
最後の(4)は，

CALLによるタスク切り替え
割り込み，例外によるタスク切り替え

に深い関係があります。CALLによるタスク切り替えは一種のサブルーチンコールです。割り込み，例外によって起動されたタスクは，一種の割り込みハンドラです。一般的にサブルーチンはRETで元のルーチンに戻り，割り込みハンドラはIRETで元のタスクに戻ります(fig.2.10)。

386では，タスクによるサブルーチンコール，割り込み処理でも同様に元に

● fig.2.10 ● タスクのネスト



戻れる機構が用意されています。それがIRETです。IRETを実行すると，EFLAGSレジスタをチェックして，

NTビットが1である

ならばタスク切り替えを起こします。つまり，現在のタスクは，ほかのタスクの実行中から呼び出されていることがわかった場合に，TSSのバックリンクフィールドで指定されるTSSセレクタでタスク切り替えを起こします。NTビットがセットされるのはJMP以外によるタスク切り替えで，IRETを使用して元に戻るのはCALLで切り替わったのか，割り込みなどで切り替わったのかを問いません。

まとめると，割り込み，例外に関連して

割り込み，例外で割り込みハンドラとしてタスクを起動できる
割り込み，例外で起動したタスクがIRETすると元のタスクに戻る

という，双方向のタスク切り替えが起こるということがいえます。また，サブルーチンコールについても

サブルーチンとしてタスクを起動できる
サブルーチンとして起動したタスクがIRETすると元のタスクに戻る

ということがいえます。

□ Busyビットの役割

JMP，IRET以外によるCALL，割り込みなどによるタスク切り替えでは，切り替わる前のタスクにあるTSSディスクリプタの**Busyビット**が1にセットされます。それ以外では0です。これは，CALL，割り込み，例外によって「切り替えられた」次のタスクへ切り替わるのが，そのタスクから切り替わった新しいタスクから戻るときに限るためです。たとえば，タスクAからタスクBにCALL/割り込み/例外で切り替わったら，次にタスクAに切り替わるのはタスクBからIRETするときで，それ以外では正しくありません。そのことを次に例示していきます。

タスク式割り込みハンドラのネスト

割り込みは実行中のプログラムとは無関係に生じます。そのため，割り込み処理をしている間に，また割り込まれる可能性があります。割り込み処理をするルーチンのことを**割り込みハンドラ**と呼びますが，386には割り込みハンドラとして，

プロシージャ形式
タスク形式

の2形式があります。この区別は，割り込みによってプロシージャが起動されるか，タスクが起動されるかの違いです。このふたつの違いは，その仕組みにあります。

プロシージャ形式では，割り込みが受け付けられると，割り込み前の状態に戻るために必要な最低限の情報を「スタック」に格納します。そして，IRETで復帰するときに元に戻します。それはEFLAGSレジスタやCS，EIPレジスタなどです。そのほかのレジスタは必要に応じて割り込みハンドラで保存します。ここで重要なのは，スタックを使用することです。スタックはLast In First Outで，スタックに格納されるときと逆の順番で取り出され，最後にプッシュされたものからポップされます。そのため割り込みが**ネスト**(割り込み処理中に割り込まれる)しても，EFLAGS，CS，EIPなどをスタックに，次々に格納し，処理が終了したら取り出します。ところが，タスク形式では，多重に割り込まれたときの履歴を保持しないために，割り込みのネストはできません。

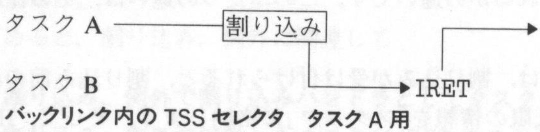
いま，割り込み番号8番で，タスクAが割り込まれたとします。そして，IDT中の8番のエントリには前に説明したタスクゲートが入っています。このとき割り込まれたタスクAのコンテキストを保存して，割り込みハンドラであるタスクBへタスクが切り替わります。さらに，このタスクに8番の割り込みがかかったとします。IDTが同じだとすると，前回と同様に8番のIDTエントリから，また，タスクBにタスク切り替えをします。しかし，これを許すと，正しい動作になりません。なぜでしょうか？

このことを理解するために，もう少し詳細な動きを見てみます。

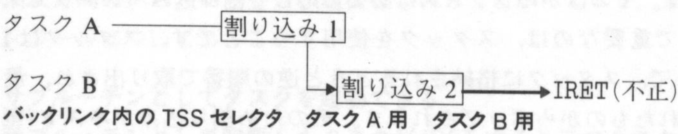
割り込みハンドラであるタスクBがさらに割り込まれたとき，タスクBのTSSのバックリンクには割り込まれたタスクAのTSSセレクタが入っていま

す。タスクBがその実行を終了してIRETする際に，タスクAのTSSでタスク切り替えを起こさなければいけません。2回目の割り込みで，またタスクBに切り替えるとすると，1回目のタスクBのコンテキストをTSSに保存します。TSSのバックリンクフィールドには，今度は割り込まれたタスクBのTSSセレクタ(結果的に自分を指す)を入れることになります。その結果，それまで入っていたタスクAのTSSセレクタは失われてしまいます。これは，1回目のタスクBはIRETによって，もはやタスクAに戻れなくなった，ということです。これを図式化すると次のようになります。

【ネストがない場合】



【ネストを許した場合】



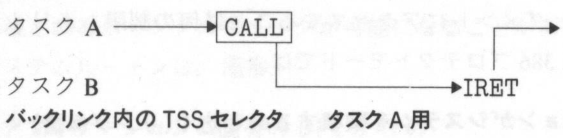
ネストを許した場合は，二度とタスクAに戻れなくなります。この例で2回目の割り込みで起動されるタスクBは，ただ単に，割り込みで中断したタスクの実行を続けるだけです。タスクBの最後で，タスクBのTSSを使ってのIRETが，終了したタスクへのタスク切り替えとなってしまい，**意味のある動作にはなりません**。こうした理由から386では，タスク式の割り込みハンドラで割り込みのネストを許していません。そのため，割り込みハンドラのタスクのTSSディスクリプタだけ，そのBusyビットをセットし，あとで調べます。386はタスク切り替えの際に，使用されるTSSディスクリプタのタイプフィールドのBusyビットをチェックし，1であれば例外が発生します。このように386では，

タスク式割り込みハンドラをもつ割り込み/例外は，同じIDでのネストを禁止

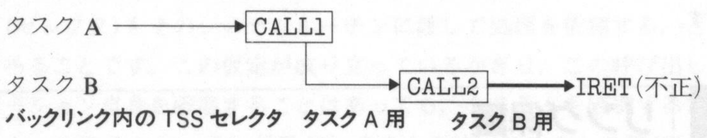
しています。

タスク式サブルーチンのネスト

同様に，タスク式のサブルーチンで，タスクAがタスクBをサブルーチンコールする場合を考えます。



タスクBが再起的にタスクBをコールする，ネストを許すと，



となり，タスクBからどこへも行けなくなってしまいます。これは，タスクでは再起的なサブルーチンコールができないことを意味しています。バックリンクがスタックにとられずに，TSSの1フィールドであるためです。こうしたことの進行を防ぐために，386のタスク切り替えでは，ビジーなタスクには切り替えられないようになっています。

以上が，タスク切り替えの仕組みと386ハードウェアサポートの内容です。その特徴をまとめると，

- **コンテキストは自動的に退避，復帰**
- **ユーザプログラムにもタスク切り替え機能を解放**
- **タスクによるサブルーチン**
- **タスクによる割り込みハンドラ**

となります。2番目と4番目はタスクゲートの効果によるものです。

3 保護

8086時代には，セグメントにアクセスすることに何の制限もありませんでした。それに対して386プロテクトモードでは，

- **アプリケーションがシステムを破壊するのを防ぐ(リング保護)**
- **アプリケーションが暴走しようとするのを発見する(タイプ保護)**

ことが可能です。こうした保護には，セグメントディスクリプタの情報が活用されます。

3.1 リング保護

リング保護の仕組みは簡単にいえば,「セグメントに設定された特権レベルよりも低いレベルではアクセス不可能」というものです。

特権レベルは0から3の値で，値が大きいほど低い特権レベルになります。セグメントに設定される特権レベルは，セグメントディスクリプタのDPLフィールドに書かれます。CPUの現在の特権レベルはCPLといって，一部の例外を除いて，現在実行中のコードセグメントのセレクタのRPLの値が使われます。

□データセグメントの読み書きとコードセグメントの読みによるリング保護

リング保護にパスするためには，アクセスするセグメントのセグメントディスクリプタのDPL，アクセスするセグメントのセレクタのRPL，そしてCPLが次の条件を満たさなければなりません。

DPL ≧ RPL しかも DPL ≧ CPL

いい換えれば，

CPL と RPL のうち特権レベルの低いほうが，DPL と同じか高い特権レベルをもたないとアクセスできない

ということになります。

RPL を使ってチェックを行う理由のひとつに，アプリケーションからシステムへ渡されるポインタのチェックが可能になることがあります。たとえば，あるシステムルーチンは，通常，

アプリケーションに固有の DPL が 3 の
セグメントを指す(TI ビット＝1)セレクタ

をもらって処理するとします。アプリケーションが，自分のセグメントへのポインタ(セレクタ)をそのシステムルーチンに渡して処理を依頼する，というのはよくあることです。この仮定が成り立っているかぎり，この呼び出しが，アプリケーション自身を破壊することはあっても，システムを破壊することはありません。システムルーチンがアクセスするのは，システムのアドレス(グローバル)空間とは独立した，アプリケーションの(ローカル)空間だからです。ところが，たちの悪いアプリケーションが，

DPL が 0 のシステムのセグメントを指す(TI ビット＝0)セレクタ

を，このシステムルーチンに渡したとします。もし，このことに気づかなければ，このルーチンはこのシステム用のセグメントを操作してしまい，アプリケーションのせいでシステム全体を危機に陥れる可能性が十分にあります。このルーチンでは，渡されるポインタが必ずアプリケーションのセグメントへのものであることがわかっているので，渡されたセレクタをシステムルーチンで使う前に，セレクタの RPL をアプリケーションの CPL(3) に合わせておけば，このセレクタをセグメントレジスタに読み込んだ際に，RPL が 3 でシステム空間を触ろうとしていることがわかります。そのため，このリング保護のルールによって呼び出しを拒否できます。

こうした使い方をすると，RPL は「要求者の特権レベルである」というのがよくわかるのではないでしょうか。

◻コードセグメントでのリング保護

同じか低いDPLのコードセグメントへの移行

コードセグメントを変更するのは，FAR JMPやFAR CALLですが，このときにリング保護がチェックされます。コードセグメントのセレクタを指定してのFAR JMPやFAR CALLが許可されるのは，通常のコードセグメントでは，

DPL = CPL しかも DPL ≧ RPL(通常)

のときだけです。通常のコードセグメントでは，より高いDPL，より低いDPLをもつコードセグメントを実行することはできません。

一方，DPLがCPLより高くても，「セグメントを変えての実行」を許す場合があります。それは，コードセグメントディスクリプタにある**コンフォーミング**(Conforming)ビットが立っている場合です。

コンフォーミングビットはコードセグメントのディスクリプタにしかありません。conformというのは，「規則に従う，形に沿う」などの意味です。コンフォーミングビットが立っているコンフォーミングセグメントは，セグメント自身のDPLよりも低いCPLからの実行の移行を許可し，RPLの値は無視します。

DPL ≦ CPL(コンフォーミング)

こうしたコンフォーミングセグメントへ実行を移しても

CPLは高い特権レベルになることはなく，前のまま変わらず

「同じ特権レベル」です。また，コードセグメントからの定数の読み出しでも，コンフォーミングセグメントに対しては，

セグメント自身のDPLよりも低いCPLからの読み出しを許可

して，結果的にすべてのCPL，RPLからの読み出しを許可します。

より高いDPLのコードセグメントへの移行

CPLよりも高いDPLをもつコードセグメントに移るには，**コールゲート**と呼ばれる機構を利用します。

ゲートの概念は，セグメントの出入口ということです。ゲートは，ただ単に出入口というだけではなく，そのゲートが担当するセグメントへ実行を移行しようとした際に，いろいろなチェックも行います。

ゲートはディスクリプタです。ゲートにはコールゲートのほかにも，タスクゲート，割り込みゲート，トラップゲートがあります。ここでは，**コールゲートディスクリプタ**について説明をします。

コールゲートディスクリプタには，

(1)飛び先のセグメントのセレクタ
(2)飛び先のセグメント内オフセット
(3)実行移行の際のスタックコピーの量
(4)ゲートディスクリプタのDPL(ディスクリプタ特権レベル)

などのフィールドがあります。

(1)で飛び先のコードセグメントディスクリプタを得て，さらに(2)で最終的な飛び先のリニアアドレスを確定します。(3)は，特権レベルが移行するときに，使用するスタックも別なものになるため，飛び先で必要なスタックの中身をコピーする，そのデータ量を記述します。

コールゲートを使用できるCPL，RPLは，データアクセスの際と同じで，

コールゲートDPL ≧ CPL しかも コールゲートDPL ≧ RPL

です。コールゲートのDPLとは，いわば「ゲートの敷居の高さ」です。

この条件が満たされたとき，実行を移行しようとしているコードセグメントのセグメントディスクリプタのDPLとCPLとを比較してみると，JMPとCALLとではチェックが少し異なっています。

JMPでは，

コードセグメントディスクリプタDPL = CPL(通常)
コードセグメントディスクリプタDPL ≦ CPL(コンフォーミング)

であるならば許可されます。

コールゲートを通じてのJMPは，コンフォーミングセグメントでないと，

コールゲートなしの場合と同じように，同じ特権レベル内での移行しかできません。

一方，CALLでは，

コードセグメントディスクリプタDPL ≦ CPL

ならば許可されます。CALLの場合は，コンフォーミングセグメントでなくても，CPLと異なるDPLのセグメントに移行できます。また，新しいCPLは飛び先のCSセレクタのRPLにセットされます。

異なるリングレベルから戻る

さて，FAR CALLで移行したコードセグメントからFAR RETで戻るとします。スタックに積まれていた元のCS：EIP，SS：ESPが復帰します。しかし，このとき，現在の(飛び先の)コードが悪さをしているかもしれないので，これらのセレクタやポインタがいろいろとチェックされます。まず，

元のCSのRPL ≧ CPL

を満たさなければならず，高いCPLにFAR RETすることはありません。

ここでFAR JMP，FAR CALLの条件が再度チェックされます。さらに，スタックが調整され，DS，ES，FS，GSの内容がチェックされます。これらのセレクタが復帰先のCPLよりも高いDPLをもつセグメントを参照する場合，INDEX＝0，TI＝0のヌルセレクタにして，低い特権レベルのコードが高い特権レベルのセグメントにアクセスしないようにします。セグメントに特権レベル上アクセスできるかどうかのチェック，つまりDPLのチェックは，そのセグメントのセレクタをセグメントレジスタへ読み込むときに行っています。コールゲートを利用したFAR CALLで呼ばれた，高い特権レベルのコードで設定されたセレクタを，セグメントレジスタに残したまま低い特権レベルに移行(FAR RET)すると，セグメントレジスタ内のセレクタが指すセグメントの特権レベルのチェックの機会が失われてしまい，その結果，低い特権レベルから高い特権レベルのセグメントが参照できるようになってしまうのです。ただし，コンフォーミングセグメント(コードセグメント)を参照している場合は，コン

フォーミングビットを，そうした低いレベルから高いレベルへの参照を許すことの表明と解釈して，そのセグメントレジスタはそのままにしておきます。

3.2 読み書き保護

セグメントディスクリプタのところで説明しているように，セグメントにはタイプがあります。セグメントレベルの保護ではリング保護のほかに，タイプによる読み書き制限が行えます。

コードセグメント(Eビットが1)であれば，読み出しは可能ですが，書き込みはできません。また，Rビットを0にすることで読み出しを禁止できます。また，データセグメント(Eビットが0)であれば，読み書き可能です。さらに，Wビットを0にすることで書き込みを禁止できます。

4 ページング

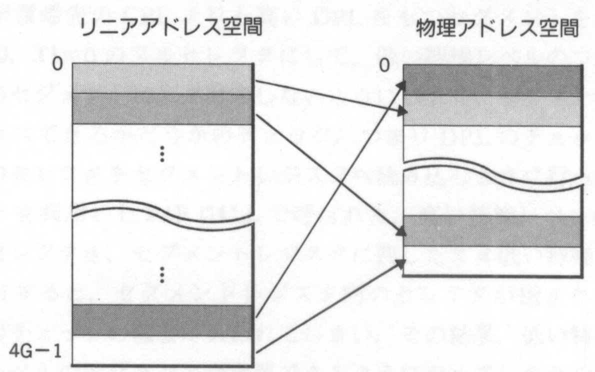
4.1 ページングの概要

ページングは，386 で追加された新しい機能です。プロテクトモードにおいて，セグメンテーション機能と同居することができます。また，プロテクトモードの一機能である仮想 86 モードでも使用できます。

ページングはセグメンテーションによって生成される 32 ビットリニアアドレス空間を，**ページ**と呼ばれる 4K バイトの領域に細分し，その 1 ページを 32 ビット物理アドレス空間の，やはり 4K バイトというサイズのページフレームにマッピングします(fig.2.11)。したがって，ページング機能によって

リニアアドレス空間と物理アドレス空間が分離

● fig.2.11 ●ページング機能



されます。たとえば，プログラムのポインタCS：EIPから生成されるリニアアドレスは，ページングが無効であれば物理アドレスを示しますが，ページングが有効なときは物理アドレスとは無関係にマッピング次第で変わります。

また，ページマッピングには属性があります。この属性によって，物理メモリとディスクとをやりとりする仮想記憶をページ単位で実現したり，システム領域にアクセスしようとするユーザプログラムを，ページ単位に拒否したりできます。こうした技術は，ワークステーションで頻繁に使用されています。

4.2 ページ変換の仕組み

まず，386のページング機能は，

CR0レジスタのPGビットが1

のときに有効です。CR0レジスタへの値の代入はリング0でのみ実行できる**特権命令**(MOV)で行います。

セグメンテーション機能によって，プログラムレベルで指定するポインタ，セレクタ：オフセットが変換され，リニアアドレスが生成されます(かっこ内はビット数)。

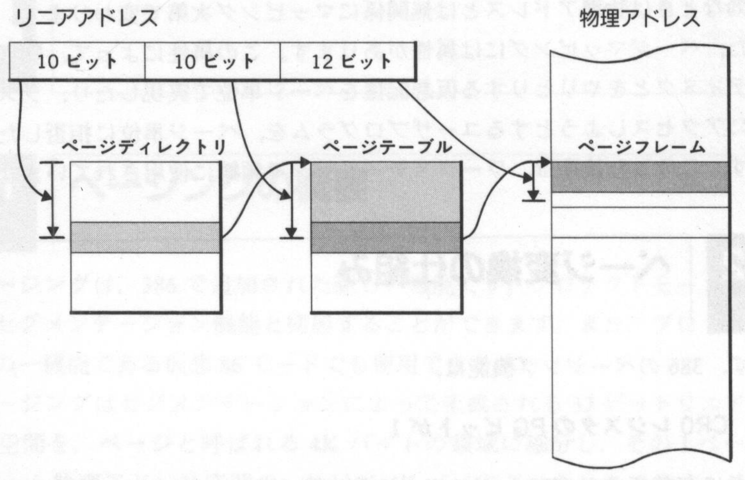
　　セレクタ(16)　　　　　：オフセット(32)
→　セグメントベース(32)：オフセット(32)
→　リニアアドレス(32)

ページングが有効であると，32ビットリニアアドレスを別解釈して

→　ページディレクトリインデックス(10)
　　：ページテーブルインデックス(10)　：オフセット(12)
→　ページフレームアドレス(20)　　　　：オフセット(12)
→　物理アドレス(32)

と変換されていきます(fig.2.12)。

● fig.2.12 ● ページ変換の仕組み



□ページディレクトリ

10ビットのページディレクトリインデックスは，ページディレクトリに対するインデックスです。ページディレクトリは1024エントリのページテーブルへの32ビットポインタの配列で，サイズは4Kバイトです。そのベース物理アドレスを，CR3レジスタのページディレクトリベースフィールド(PDBR：20ビット)が指しています。ページディレクトリはページアラインされます。マルチタスクのところで説明したように，CR3すなわちPDBRは，TSSに格納されるタスクの属性のひとつです。タスクごとに別のページディレクトリをもつことができ，異なるページ変換を行えます。

ページディレクトリインデックスによって選ばれた**ページディレクトリエントリ**(PDE)のフォーマットは，次ページ上の表のようになっています(かっこ内はビット幅)。

4Mバイトといっているのは，PDEが指すページテーブルが変換するすべての領域のサイズです。PDEが変換するページ(4Mバイト)のどこかにアク

ページフレームアドレス(20)	ページテーブルベース
システム利用可能(3)	システムに開放
Accessed (A)ビット	1：4Mバイトにアクセスした 0：プログラムでクリア
User/Supervisor (U/S)ビット	1：4Mバイトがユーザページ 0：4Mバイトがスーパバイザページ
Read/Write (R/W)ビット	1：4Mバイトがすべての特権レベルで書き込み可 0：4Mバイトが特権レベル3で書き込み不可
Present (P)ビット	1：ページテーブルがメモリ上に存在 0：ページテーブルが不在

セスされると，CPUはPDEのAビットを自動的にセットします。CPUによってクリアされることはありません。

回ページテーブル

ページテーブルインデックスは，選択されたPDEが指すページテーブルのインデックスです。ページテーブルインデックスによって選ばれたページテーブルエントリ(PTE)のフォーマットは，次のようになっています(かっこ内はビット幅)。

ページフレームアドレス(20)	ページフレームベース
システム利用可能(3)	システムに開放
Accessed (A)ビット	1：4Kバイトにアクセスした 0：プログラムでクリア
Dirty (D)ビット	1：4Kバイトに書き込み 0：プログラムでクリア
User/Supervisor (U/S)ビット	1：4Kバイトがユーザページ 0：4Mバイトがスーパバイザページ
Read/Write (R/W)ビット	1：4Kバイトがすべての特権レベルで書き込み可 0：4Kバイトが特権レベル3で書き込み不可
Present (P)ビット	1：ページフレームがメモリ上に存在 0：ページフレームが不在

4Kバイトといっているのは，PTEが指すページフレーム領域のことです。PTEが変換するページ(4Kバイト)のどこかにアクセスされると，CPUはPTEのAビットを自動的にセットします。また，PTEが変換するページのどこかに書き込みをすると，CPUはPTEのDビットを自動的にセットします。CPUによってクリアされることはありません。

4.3 ページレベルの保護

リニアアドレスが実際にアクセスする物理アドレスを，次のような流れで求めました。

①リニアアドレス(32)をページディレクトリインデックス(10)，ページテーブルインデックス(10)，オフセット(12)に分割

↓

② PDBRとページディレクトリインデックスによってPDEを得る

↓

③ PDEとページテーブルインデックスによってPTEを得る

↓

④ PTEとオフセットによって物理アドレスを得る

リニアアドレスを生成した時点で，セグメンテーションレベルのアクセス制御にすべてパスしているわけですが，さらにページングレベルでのアクセス制御を受けます。

PDE，PTEのU/Sビットの組み合わせと，最終的なページのU/S属性は，次の通りです。

PDE	PTE	結果U/S
0	0	0
0	1	0
1	0	0
1	1	1

PDEのU/SとPTEのU/SのANDです。つまり，PDEとPTEの両方のU/Sビットがセットされないとユーザページにならず，それ以外ではリング3からのアクセスはできません。PDE，PTEのR/Wビットの組み合わせと，最終的なページのR/W属性は，次の通りです。ただし，最終的なU/S属性が1であるユーザページに対してだけ，意味のある属性です。

PDE	PTE	結果 R/W
0	0	0
0	1	0
1	0	0
1	1	1

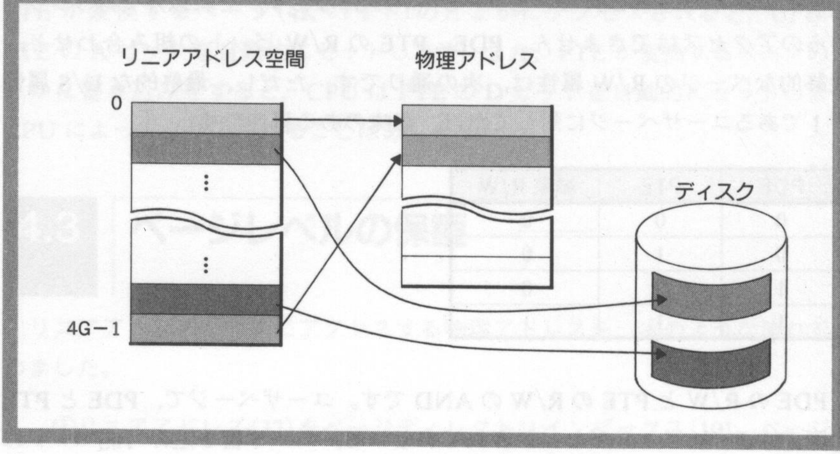
PDEのR/WとPTEのR/WのANDです。ユーザページで，PDEとPTEの両方のR/Wビットがセットされていないと，ユーザ書き込み可能ページになりません。

4.4 仮想記憶

PDE，PTEのPビットによって**ページ単位の仮想記憶**を行えます(fig.2.13)。Pビットが0のとき，32ビットのPDE，PTEの残りの31ビットはシステムで自由に使えます。たとえば，ページフレームをメモリからディスクに送り，その場所をこの31ビットに書いておきます。ディスクに追い出されたページにアクセスするとページ不在の例外が生じるので，起動される割り込みハンドラはディスクから空きページフレームにページを読み込みます。空きページがない場合には，アクセス頻度の少ないページから破棄，またはディスクへのスワップを行います。

ページごとのアクセス頻度を測るためには，PDE，PTEにAビットを利用します。システムは適当な間隔で，PTEのAビットが立っているかどうかを見て，立っていればそのページのアクセス頻度値を増やし，Aビットをクリアします。これを繰り返すことによって，ページのアクセス頻度統計をとることができます。

● fig.2.13 ●仮想記憶



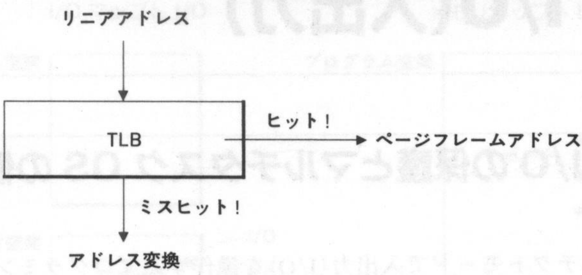
また，アクセス頻度の低かったページを破棄するかどうかは，Dビットを利用します。書き込みがあったページのPTEのDビットはセットされているので，そのようなページはディスクにページアウトして保存します。Dビットがセットされていなければ，メモリにページインされたときのままですから，ディスクに残っているページは，破棄してもかまいません。

以上は，PTEのPビットによるページフレームのディスクとのやりとりです。PDEのPビットを使うと，ページテーブルのメモリ中の有無を指定できます。PDEのPビットを0とした場合，該当する4Mバイトの領域はリニアアドレス空間として無効であるか，あるいはページテーブル自身がディスクにページアウトされているか，どちらの使い方もできます。前者は不必要なページテーブルをもたずにページ変換テーブルを小さくするために役立ちます。

4.5 TLB

セグメンテーションでは，セレクタをセグメントレジスタにロードして保護

● fig.2.14 ● TLB



のチェックが問題ないと，セグメントディスクリプタは「見えないレジスタ」にロードされ，セレクタが変更されないかぎり保持されます。これはセグメントディスクリプタセレクタのキャッシュで，セグメントディスクリプタというようにアドレス確定の手続きを1段階増やしても効率が落ちないようにするためには，重要なメカニズムです。

ページングでも，一度ページ変換したリニアアドレスを二度目に変換するときには，効率よく変換するように，TLBと呼ばれるページ変換キャッシュをもちます。

TLBは，最近変換されたページを32個もちます。変換しようとするリニアアドレスのページがTLB内にあれば，ページディレクトリやページテーブルを引かずに，対応するページフレームアドレスを高速に得ることができます(fig.2.14)。ほとんどのメモリアクセスで，TLBにヒットします。

5 プロテクトモードのI/O(入出力)

5.1 I/Oの保護とマルチタスクOSの信頼性

386のプロテクトモードで入出力(I/O)を操作するプログラミングを行うには，8086のときのI/O操作に比べて，理解しなければならないことが若干あります。

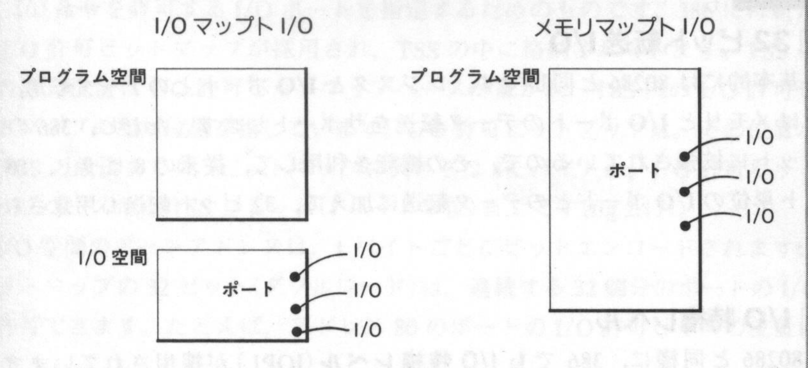
まず，8086ではI/O命令が何の制限もなく実行されますが，386プロテクトモードではI/O操作がメモリアクセスと同様に保護の対象となります。

I/Oは，システム全体で共有する場合が多くあります。個々のタスクから同じ種類のI/O要求がくるので，何の制限もなくそれらのI/O操作を許可すると，ひとつのタスクの誤りが全体に波及するという危険があります。そのため，386プロテクトモードでは，主に個々のタスクのI/O要求を調停するシステムプログラムがI/O操作を行います。たとえば，I/O操作をシステムコールが一括して行い，それ以外のアプリケーションレベルからのI/O操作を禁止します。このおかげで，386プロテクトモードでのマルチタスクOSでは，行儀の悪いアプリケーションが直接I/Oを操作しようとしても，それは拒否され，ほかのタスクの安全性が保たれます。

5.2 I/Oポートの置き場所

一般的に，I/Oを行うためには，決められた，あるいは決めたアドレスにプログラムをアクセスします。たとえば，外からのデータが，あるアドレスのメモリに用意されていれば，それを読みます。また，外へ送るデータをメモリに書き込みます。このようなI/Oのための記憶域を**I/Oポート**と呼びます。

● fig.2.15 ● I/OマップトI/OとメモリマップトI/O



このI/Oポートを置く場所は，大きくふたつに分類されます。ひとつはI/Oポートを

プログラムのアドレス空間とは独立したI/O空間におく

方法で，**I/OマップトI/O**といいます。I/OマップトI/Oの場合，I/Oポートはプログラムのアドレス空間上にないため，通常命令ではアクセスできず，I/O専用命令を使用します。8086はI/OマップトI/Oを使用できます。

もうひとつは，I/Oポートを

プログラムのある領域にマッピングする

方法で**メモリマップトI/O**といいます。I/OマップトI/Oと違って，通常のメモリアクセスの要領でI/Oポートを読み書きできます(fig.2.15)。

次の項では，386プロテクトモードのタスクのI/Oについて，どのようにしたらI/O命令の実行を制限できるか，また許可できるかについて説明していきます。

5.3 I/O マップト I/O

回 32 ビット転送 I/O

基本的には 80286 と同じです。レジスタと I/O ポートとのデータ転送，またはメモリと I/O ポートのデータ転送をサポートします。ただし，386 の 32 ビットに拡張されているので，その機能を利用して，従来の 8 ビット，16 ビット単位の I/O ポートとのデータ転送に加えて，32 ビット転送も用意されます。

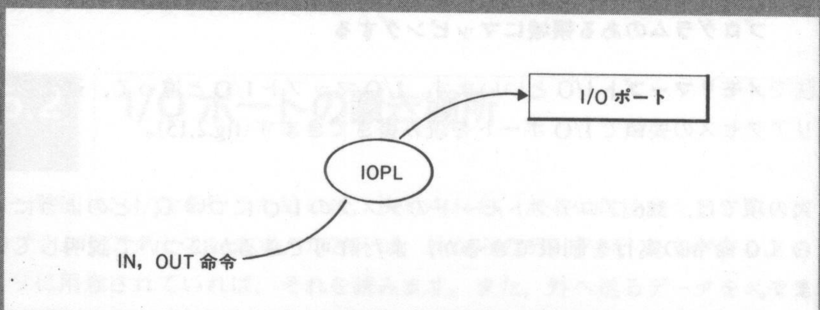
回 I/O 特権レベル

80286 と同様に，386 でも **I/O 特権レベル**(IOPL)が採用されています。IOPL とは I/O 命令を使用するための特権レベルで，EFLAGS レジスタの 1 フィールドです。EFLAGS はタスク個別の属性なので IOPL は各タスクに設定できます。IOPL を変更する命令は，I/O 命令を実行すると，現在の特権レベル CPL と現在の IOPL が比較され，CPL が IOPL よりも低いレベルであればその I/O 命令は拒否されます(fig.2.16)。つまり，IOPL は，

タスク中で I/O 命令を発行できる特権レベルを制限

するということです。

● fig.2.16 ● IOPL に制限される I/O 命令



□ I/O 許可ビットマップ

I/O 許可ビットマップは，CPL が IOPL よりも低いレベルであった場合でも，I/O 命令を許可する I/O ポートを指定するためのものです。386 には新たに I/O 許可ビットマップが採用され，TSS の中に格納されています。TSS の中の固定位置に I/O 許可ビットマップベースが置かれ，TSS 内の I/O 許可ビットマップの相対位置を指しています。I/O 許可ビットマップは，その位置から TSS の最後までで，ビットマップの大きさは 8K バイトか，I/O 許可ビットマップベースから TSS リミット(1 バイト単位)までです(fig.2.17)。

I/O 空間のポートアドレスは，1 バイトごとにビットエンコードされます。ビットマップの 32 ビット(ダブルワード)は，連続する 32 個分のポートの I/O が許可できます。たとえば，アドレス 80 のポートの I/O 許可ビットの位置は，

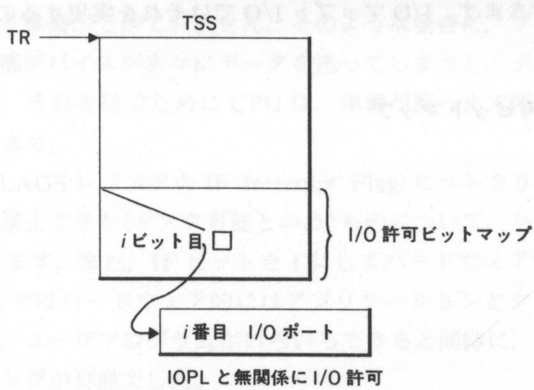
$$80 = 32 \times 2 + 16$$

であることから，ビットマップの 3 番目のダブルワードの第 16 ビットです。

このビットが「セット」されていればアドレス 80 への I/O 命令は IOPL のチェックを受け，セットされていなければ受け付けられます。

実際には IOPL のチェックを最初に受けますが，プログラムレベルでみれば，

● fig.2.17 ● TSS 中の I/O 許可ビットマップと許可ビット



I/O 許可ビットマップで，

1にセットされているポートはIOPLによる特権レベルチェックを受け，
0にセットされているポートはすべてのI/O命令を許可

しているのと同じです。つまり，I/O 許可ビットマップは，I/O 命令が

特権レベルのチェックを受けるかどうかポートごとに指定

するためのデータと考えてもかまいません(仮想86モードでは若干異なる)。I/O 許可ビットマップは，すべてのポートに用意される必要はありません。ビットマップをTSSの最後に押しやって，ビットマップ後半をもたずにすますことができます。TSSからはみでた分についてのビットマップは1とみなされ，そのポートに対するI/O命令は特権レベルのチェックを受けます。

回メモリマップトI/O

メモリマップトI/Oでは通常のメモリアクセスと同様の保護を受けます。セグメンテーションならば，タイプやリングレベルなどによるアクセス制限が適用されます。ページングならば，読み書き属性やユーザ/スーパバイザ属性によるアクセス制限が適用されます。

このように386プロテクトモードでは，システムと比べれば信頼性の低いアプリケーションレベルからのI/O操作を禁じて，タスク間の独立性をより高めることができます。I/OマップトI/Oではそれを実現するのが，

- **IOPL**
- **IO許可ビットマップ**

です。

6 割り込みと例外

6.1 特権レベルで制限を受ける割り込み/例外処理

IOPLは名前こそI/O特権レベルですが，外部割り込みのマスク(IFビットクリア)の制限にも使用されます。IOPLに影響されることを「IOPLセンシティブ」といいます。また，ソフトウェア割り込み(トラップ)やEFLAGSレジスタのIOPLフィールドの変更は，特権レベルによって制限を受けます。

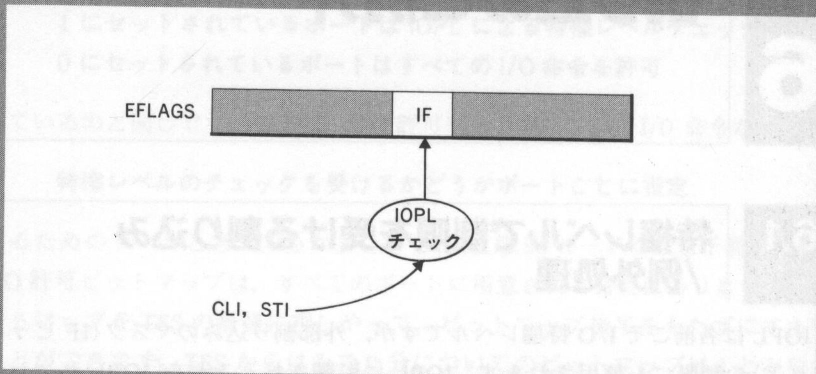
◻ハードウェア割り込み禁止/許可

ハードウェア割り込みは，CPUに接続されているデバイスが，CPUに何かを通知するための手段です。しかし，割り込みの種類によっては，一時的に割り込みを禁止したい場合があります。たとえば，データがI/Oポートに用意できることを，外部デバイスがCPUに通知したいとします。しかし，CPUは，まだ前回のデータをバッファに取り込んでいないか，バッファを用意できないかで受け取る準備ができていません。そのような場合に，プログラムの状況を無視して外部デバイスが次々にデータを送ってしまうと，データの取りこぼしが生じます。それを防ぐためにCPUは，準備が整うまで割り込みを禁止することができます。

8086では，FLAGSレジスタのIF(Interrupt Flag)ビットを0にして，外部割り込みのうち禁止できる(マスク可能という)ものについて，ハードウェア割り込みを禁止します。また，IFビットを1にしてハードウェア割り込みを許可します。8086ではハードウェア的にはアプリケーションとシステムの境界がなかったので，ユーザプログラムがいろいろできると同時に，その分だけ危ないプログラミングが可能でした。

386のシステムの信頼性やタスク間の独立性を保証するマルチタスクOSで

● fig.2.18 ● IFアクセス制限するIOPL

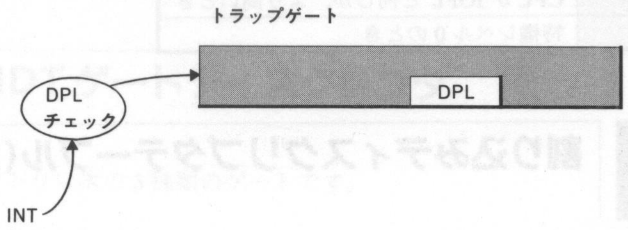


は，このようなアプリケーションレベルから割り込み禁止/許可をすることを許しません。その理由は，もし，アプリケーションから自由に割り込みを禁止できると，行儀の悪いアプリケーションが割り込みを禁止したまま無限ループに陥いった場合，そのタスクからCPUを解放するにはNMI(Non-Maskable Interrupt)，すなわちマスクできない割り込みをするしか方法がないためです。また，システムレベルで禁止した割り込みを，アプリケーションレベルで割り込み許可を許せば，システムが壊れかねません。こうした問題を解決するために，386プロテクトモードでは割り込みを，マスク/マスク解除できる特権レベルをIOPLによって制限します。つまり，ハードウェア割り込みの許可を示すIFビットを操作できるのは，IOPL以上の特権レベルのコードとなります(fig.2.18)。

□ソフトウェア割り込み(INT)の制限

8086のソフトウェア割り込みであるINT命令は，基本的にはすべて受け付けられます。386プロテクトモードでは，INT命令は例外(トラップ)を引き起こし，あとで説明するトラップゲートのDPLによるCPLのチェックを受けてから，CPLがトラップゲートのDPLよりも低ければソフトウェア割り込みを許可しません。このメカニズムによってシステムコールに，特権レベル0からしか呼べないシステムコールと，アプリケーションでも呼べるシステムコー

● fig.2.19 ● INTを制限するトラップゲートDPL



ルという区別ができます(fig.2.19)。

回フラグレジスタ(EFLAGS)変更に関する制限

8086ではPUSHF, POPFによってそれぞれ，FLAGSレジスタの内容を**スタックに格納，スタックから読み出し**を行います。386プロテクトモードでは，PUSHFD，POPFDによってEFLAGSレジスタとスタックとのやりとりを行いますが，EFLAGSのすべての内容が対象となるわけではありません。

POPFDによるEFLAGSレジスタの変更で，特権レベルの制限を受けるフィールドは具体的には，次の通りです。

VM，RF	ビット	変更されない
IF	ビット	CPL値≦IOPL値の場合(IOPLセンシティブ)
IOPL	フィールド	CPL＝0の場合のみ

IOPLフィールドは，特権レベル1から3でのPOPFDでは変更されません。POPFDによるIFビットの変更は，やはりIFビットを直接操作するSTI，LTIと同じくIOPLセンシティブです。

また，IRETによってスタックに退避していたEFLAGSをレジスタに復帰しますが，この場合もIOPLフィールドが変更されるのはCPL＝0のときだけで，IFビットが変更されるのはCPLの値≦IOPLの値の場合となります。

つまり，POPFDでもIRETでも，EFLAGSレジスタへの値の代入の際にIF，IOPLが変更されるのはそれぞれ，次の通りです。

IF	CPLがIOPLと同じか，より高いとき
IOPL	特権レベル0のとき

6.2 割り込みディスクリプタテーブル(IDT)

□割り込みベクタテーブル

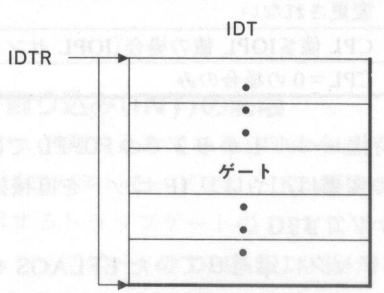
プロテクトモードでは，8086でいうところの割り込みベクタテーブルを，

割り込みディスクリプタテーブル(IDT)

という形でもっています。IDTは，割り込み/例外が生じたとき，そのIDでテーブルを引かれるという点では8086の割り込みベクタテーブルと同じ役目をもちます。IDTはひとつのセグメントで，IDTレジスタ(IDTR)によってその場所(リニアアドレス)とリミットをもちます。IDTRの変更は，LIDT命令によって行います。しかし，これは特権命令ですから，

IDTの場所とサイズは特権レベル0でしか変更できない

● fig.2.20 ● IDTの構造(含むゲート)



ことになります。IDT は 8086 の割り込みベクタテーブルと同じく 256 までの ID をもつことができますが，異なるのは IDT の 1 エントリは 8 バイトの大きさの「ゲート」ディスクリプタであるということです(fig.2.20)。

6.3 IDT ゲートディスクリプタ

IDT のエントリは次の 3 種類のゲートです。

- **割り込みゲート**
- **トラップゲート**
- **タスクゲート**

それぞれについて，解説していきましょう。

□ IDT エントリが割り込み/トラップゲートの場合

プロシージャ形式の割り込みハンドラ

受け付けられた割り込み/例外 ID に該当するエントリが，割り込み/トラップゲートである場合は，

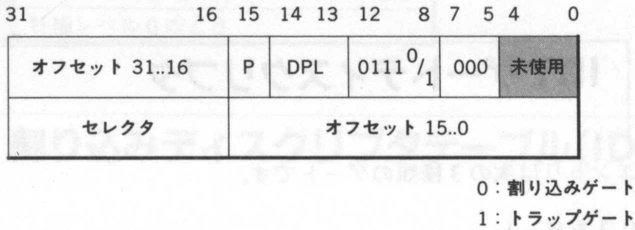
プロシージャ形式(FAR CALL 形式)で割り込みハンドラを起動

します。割り込みゲートとトラップゲートの 2 種類のディスクリプタは,「タイプが違う」または「引数の量を指定する DWORD フィールドがない」ことを除いては，コールゲートと同じフォーマットをもちます(fig.2.21)。すなわち，割り込み/トラップゲート中の P ビットは，ゲートが有効であることを示し，

- **DPL フィールドでゲートへのアクセスチェックを行い(INT 命令のみ), セレクタフィールドが割り込みハンドラのコードセグメントディスクリプタを指す**
- **オフセットフィールドがコードセグメント中の割り込みハンドラ実行開始アドレスを指す**

ことになります。

● fig.2.21 ●割り込み/トラップゲート



割り込みを受け付け，割り込みハンドラのコードセグメントディスクリプタを得ると，コールゲートを使ったFAR CALLと同様に，割り込み/トラップゲートを使って割り込みハンドラをコールします。コールの方法は，現在のリングと同じか，内側リングの割り込みハンドラを呼び，IRETで元のリングに戻ります(詳しくは，コールゲートを使ったp.61のFAR CALLの説明を参照)。

受け付けたときのスタックの振舞いも，最後にエラーコードをスタックにプッシュすることがあることを除けば，コールゲートを使ったFAR CALLと同じです。実際に割り込みハンドラへ飛ぶ前に，EFLAGSレジスタの各種フラグを操作します。CPUは，シングルステップ実行を実現するためのEFLAGSレジスタのTF(Trap Flag)ビットを0にします(前のTFはEFLAGSとしてスタックに退避)。これによって割り込みハンドラ内では，シングルステップ実行をしません。また，NT(Nested Task)ビットも0にします。これはIRETの実行でプロシージャとしてリターンするためです。使用したIDTエントリが割り込みゲートの場合は，IF(Interrupt Flag)ビットを0にし，これによって外部割り込みを禁止します。トラップゲートの場合はそのままです。つまり，このふたつのゲートの違いは外部割り込みマスクにあり，

- **割り込みゲート………IFビットをクリアして外部割り込み禁止**
- **トラップゲート………IFビットはそのまま**

となります。TF，NT フラグは割り込みハンドラの IRET で EFLAGS の復帰とともに割り込み/例外発生前の状態に戻りますが，前に説明したように，

IF フラグは，割り込みハンドラの特権レベルが IOPL と同じか，高くなければ復帰しない

ので，注意が必要です。

割り込みハンドラの特権レベル

プロシージャ形式の割り込みハンドラは，基本的に割り込み/例外を発生するときの特権レベルと同じか，高いレベルで実行されなければなりませんが，割り込みは実行されているコードの特権レベルと無関係に生じるので，通常のコードセグメントで，常にこの条件を満たそうとすれば，

割り込みハンドラはリング 0 で実行

ということになります。また，FAR CALL のところでも説明しているように，コンフォーミングセグメントを使うと，CPL がコードセグメントの特権レベルより高くても実行を許すことを利用して，たとえ割り込み/例外発生時の CPL が割り込みハンドラの特権レベルよりも高いとしても

割り込みハンドラをコンフォーミングセグメントに置く

ことによって解決されます。

回 IDT エントリがタスクゲートの場合

タスク式割り込みハンドラ

IDT エントリがタスクゲートの場合は，割り込みハンドラとして働くタスクに切り替わります。タスクゲートは起こすべきタスクの TSS セレクタをもち，必要なコンテキストとともにタスクを切り替えます。割り込みハンドラタスクは再入を防ぐため，ビジー状態で割り込み/例外が発生したタスクにネストします(タスク式割り込みハンドラの起動の流れは，本章の「2　マルチタスク」p.41 を参照。fig.2.22)。

● fig.2.22 ●タスクゲート

31 16	15	14 13	12 8	7 0
未使用	P	DPL	00101	未使用
TSSセレクタ	未使用			

補足として，割り込み/例外がエラーコードを発生した場合には，割り込みハンドラタスクの新しいスタックにエラーコードがプッシュされている，ということがあります。

6.4 割り込み/例外の種類とエラーコード

□割り込み/例外の分類

割り込み/例外のリストと，それぞれエラーコードを発生するかどうかを示しておきます。

割り込みも例外も，いったん受け付けられると，現在実行中のプログラムの流れから外れる現象です。**割り込み**はタスクの動作とは非動機なイベントで，**例外**は実行されているコードが原因で引き起こされます。つまり，割り込みと例外の違いは，

- **割り込みは，実行されているコードとは無関係に起こる**
- **例外は，実行されているコードが原因で起こる**

ことです。

割り込みには，マスク可能なINTRとマスク不可能なNMIがあります。例外には，例外を生じさせる命令が起こすもの(ソフトウェア割り込み)と，命令

を実行する条件を満たしていないときに発生するものの2種類があります。また，その検出のされ方によって，次のように3つに分類されます。

	発生時期	例外発生のタスクに戻る可能性
フォールト	命令実行前/中	あり
トラップ	命令実行後	あり
アボート	不明	なし

スタックに退避されるCS:EIPでみると，次の通りです。

	スタックに退避されるCS:EIPが指す場所
フォールト	例外を発生した命令
トラップ	例外を発生した命令の次に実行する命令
アボート	実行再開にはならない場所

また，割り込みハンドラの仕事でみると，次の通りです。

	割り込みハンドラの仕事
フォールト	命令再実行のために例外の原因の除去
トラップ	次命令実行のために例外要求を処理
アボート	破壊されたタスクからの脱出

□割り込み/例外の種類

割り込み/例外のIDとその分類，発生のおおよその原因を，次の表に示します。

ID	エラーコード	分類	原因
0		フォールト	除算エラー
1		フォールト/トラップ	シングルステップなどデバッグ
2		割り込み	NMI
3		トラップ	ブレークポイントなどデバッグ
4		トラップ	オーバフロー(INTO命令)
5		フォールト	境界チェック(BOUND命令)
6		フォールト	無効命令
7		フォールト	コプロセッサ不在

8	有	アボート	ダブルフォールト
9		アボート	コプロセッサのセグメント超過
10	有	フォールト	無効 TSS
11	有	フォールト	セグメント不在
12	有	フォールト	スタック例外
13	有	フォールト	一般保護例外
14	有	フォールト	ページ例外
15		予約 ID	
16		フォールト	コプロセッサ演算エラー
17-31		予約 IDs	
32-255		割り込み/トラップ	マスク可能外部割り込み/INT *n* 命令

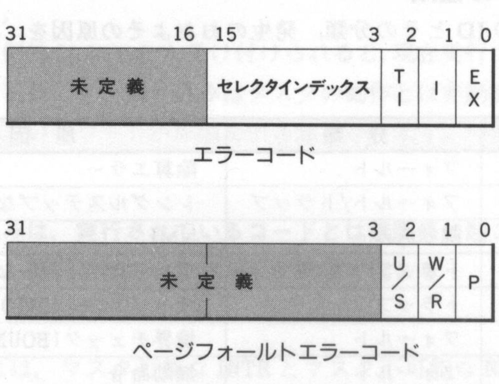
INT*n* 命令は，割り込み ID として 0 から 255 までを使用できます。

□エラーコード

一部の割り込み/例外はスタック上にエラーコードをプッシュします。**エラーコード**は，例外を処理するために必要な追加の情報を提供します(fig.2.23)。割り込み/例外 ID が 8 の場合，エラーコードは 0 です。

割り込み/例外 ID が 10，11，12，13 の場合，32 ビットのエラーコードの下

● fig.2.23 ● 2種類のエラーコード



位ワードフォーマット(上位ワードは0)は，セレクタとよく似ていて，次の4つのフィールドからなります。

セレクタインデックス(14ビット)	原因となったセレクタ
TI ビット	I=0のとき 1：インデックスがLDT中のゲートを指す 0：インデックスがGDT中のゲートを指す
I ビット	1：インデックスがIDT中のゲートを指す 0：インデックスがLDT/GDT中のゲートを指す
EX ビット	1：例外がプログラム外のイベントの結果

割り込み/例外IDが14の場合，32ビットのエラーコードの下位3ビットのフォーマット(ほかは未定義)は，次の通りです。

U/S ビット	1：ユーザモードのアクセスでページフォールト 0：スーパバイザモードのアクセスでページフォールト
W/R ビット	1：書き込みアクセスでページフォールト 0：読み出しアクセスでページフォールト
P ビット	1：ページレベル違反が原因 0：ページ不在が原因

7 仮想 86 モードのすべて

80286 でもマルチタスクを実現しているので，この点だけをとってみると，386 は 80286 とあまり変わらない CPU のように見えます。しかし，386 には，80286 と比べて決定的に違う点があるのです。それが**仮想 86 モード**です。

7.1 仮想 86 モードとは

80286 の欠点のひとつに，「MS-DOS をサポートするためには，リアルモードを使う」ということがあります。それは，たとえプロテクトモードを利用した安全で信頼性の高い OS を使用していても，MS-DOS を利用するためには，いったんリアルモードにならなければならないということです。リアルモードはシステム保護を受けられないモードなので，80286 の OS で MS-DOS をサポートすることは，システムの信頼性を下げることを意味するのです。また，リアルモードという別モードに移行するので

MS-DOS 環境↔プロテクトモード OS

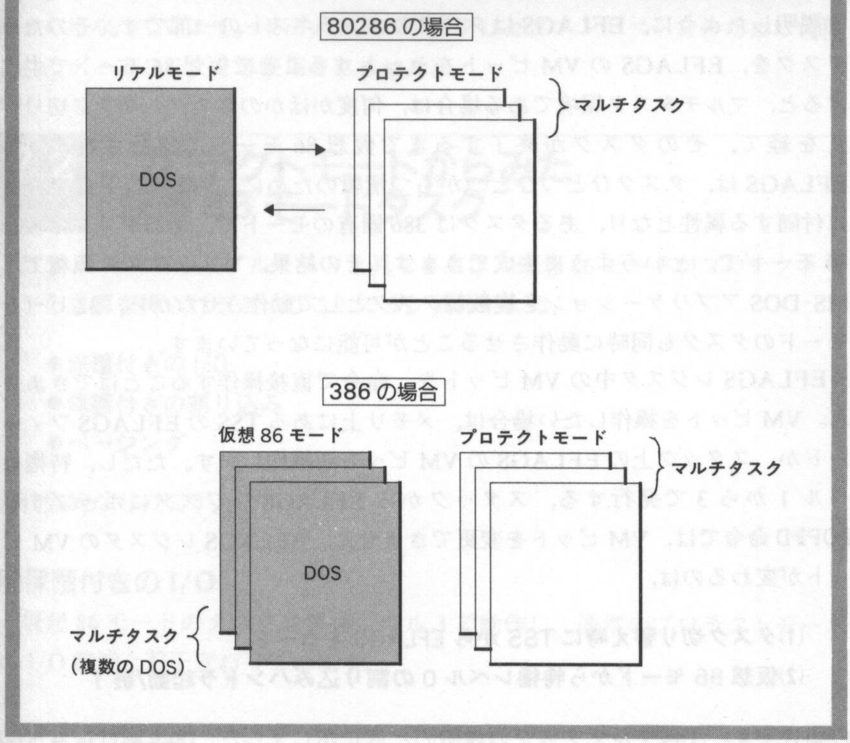
という二重の環境となってしまい，MS-DOS が動作しているときにプロテクトモード OS は眠り，またその逆に，プロテクトモード OS が動作しているときに MS-DOS は眠ることになります。たとえば，プロテクトモード OS のアプリケーションで通信を行っているとき MS-DOS を起動すると，その時点でプロテクトモード OS のアプリケーションは OS と同時に止まります(fig.2.24)。

ところが 386 では，この点が大きく違っています。一言でいうと，386 では，

8086 互換で動作するタスクのモードがある

ということです。この結果，プロテクトモード OS の 1 タスクとして 8086 プ

● fig.2.24 ● 80286/386のDOSサポートと仮想OS



ログラムが動作します。MS-DOSは，そのもっとも典型的なものです。386では，80286のようにリアルモードに移行する必要がないのです。プロテクトモードOSのタスクとしてMS-DOSが動作することの利点は，

- **システムはプロテクトモードのままなので安全**
- **マルチタスク環境で複数のDOSアプリケーションを同時に起動可能**

ということが代表的なことです。

このような386のタスクモードを**仮想86モード**と呼びます。EFLAGSレ

ジスタの中のVirtual 8086 Mode(略してVM)ビットがセットされていると，そのタスクは仮想86モードで実行されます。本章の「2　マルチタスク」(p.41)で説明したように，EFLAGSはタスクのコンテキストの一部です。そのため，タスクを，EFLAGSのVMビットをセットすることで仮想86モードで生成すると，マルチタスク環境である場合は，何度かほかのタスクへタスク切り替えを経て，そのタスクが終了するまで仮想86モードで実行されます。EFLAGSは，タスクひとつひとつがもつ情報のために，VMビットはタスクに付随する属性となり，あるタスクは386固有のモードで，あるタスクは仮想86モードで，というように生成できます。その結果，マルチタスク環境で，MS-DOSアプリケーションを複数個タスクとして動作させながら，32ビットモードのタスクも同時に動作させることが可能になっています。

EFLAGSレジスタ中のVMビットを，命令で直接操作することはできません。VMビットを操作したい場合は，メモリ上にあるTSSのEFLAGSフィールドか，スタック上のEFLAGSのVMビットを操作します。ただし，特権レベル1から3で発行する，スタックからEFLAGSレジスタにポップするPOPFD命令では，VMビットを変更できません。EFLAGSレジスタのVMビットが変わるのは，

(1)タスク切り替え時にTSSからEFLAGSをロード
(2)仮想86モードから特権レベル0の割り込みハンドラ起動/終了

の場合です。(1)はマルチタスクの説明のときに示しました。(2)は割り込みハンドラが「特権レベル0」のときにだけ，EFLAGSレジスタの退避/復活時にVMビットを，それぞれ1から0，0から1に変更できます。つまり，仮想86モードから起動される特権レベル0の割り込みハンドラは，仮想86モードではなく，通常のプロテクトモードで実行されます。

386はタスク切り替えを起こして，EFLAGSがロードされると，VMビットをチェックし，セットされていた場合はコードを8086コードとして解釈します。また，セットされていなかったら，コードを386コードとして解釈します。仮想86モードのタスクは，必ずレベル3の特権レベルで動作します。

仮想86モードはタスクのモードなので，仮想86モードで実行されるプログラムはひとつのタスクです。そのためTSSをもちます。仮想86モードは，ほ

とんど 8086 なので 386 のレジスタもリアルモードのように使用されます。そのため，TSS に保存されるレジスタ群の内容はリアルモードと同じ意味をもつものになっています。セグメントレジスタはセグメントベースのために使用され，ディスクリプタは使用されません。

7.2 プロテクトモードからみた仮想 86 モードタスク

仮想 86 モードがリアルモードと異なる点は，仮想 86 モードがプロテクトモードの恩恵を得られるということです。具体的には，

- **保護付きの I/O**
- **保護付きの割り込み**
- **ページング**

が代表的なものです。

回保護付きの I/O

仮想 86 モードのタスクは特権レベル 3 で動作し，通常のプロテクトモードの I/O 機能と若干変わっています。

I/O マップト I/O

通常のプロテクトモードでは，IOPL に影響された IN，OUT などの実際のポートとデータのやりとりをする I/O 命令は，仮想 86 モードでは IOPL の影響を受けません。このことを，

I/O 命令は仮想 86 モードで IOPL センシティブな命令ではない

といういい方をします。仮想 86 モードにおける I/O 命令は，タスクの TSS 中の I/O 許可ビットマップによってのみ制限を受けます。IOPL によるチェックがない分，制限は単純化され，I/O 許可ビットマップのビットが，

- **0 のポートへの I/O 命令は許可**

●1のポートへのI/O命令は禁止

となります。

仮想86モードで動作する8086プログラムは，実際の8086では物理的なI/Oポートに直接アクセスします。しかし，プロテクトモード下の386では，仮想86モードタスクにそのようなことを許すことはできません。とはいっても，8086で動作しているのと同じI/O操作を提供しなければなりません。そのために，I/O許可ビットマップによるI/O命令禁止機能を利用して，システムがI/O操作をエミュレートします。8086プログラムがI/O操作を行うと，I/O許可ビットマップによって拒否され，例外を生成します。例外を処理するシステムプログラムは，ほかのタスクのI/O操作との調停をとりつつ，8086プログラムのI/O操作を実行します。こうしたエミュレーションは，8086プログラムを1タスクとして動かしたときに，その発行するI/O命令がシステムの正しさを保証できなくなるような場合に有効です。

メモリマップトI/O

仮想86モードでは，ページングを使用できます。仮想86モードのタスクのアドレス空間(見かけ上8086と同じ)は，ページングによって物理ページにマッピングされます。メモリマップトI/Oでは実際のI/Oポートは物理アドレスに置かれるわけですから，I/Oポートを使用する仮想86モードタスクは，そのポートを含む物理ページをアドレス空間のどこかにマッピングして，I/Oポートに通常のメモリの読み書きでアクセスします。

その際，ページ属性を利用すれば，仮想86モードタスクのI/O操作をOSなどのシステムプログラムが,「横取り」する機会が生じます。このことを利用すれば，仮想86モードタスクのI/O操作をシステムプログラムがシミュレートすることができます。仮想86モードで動作する8086プログラムは，直接操作しているつもりでも，実際はシステムプログラムが代行していることになります。I/Oのようなシステム共通の資源を，仮想86モードタスクには8086で動作しているのと同じ効果をもたらしつつ，仮想86モードの1タスクが自由にアクセスすることを防ぎます。たとえば，I/Oポートを読み出し専用ページ

に置いた場合，ポートへの書き込み時に，I/O ポートを置いたページをページ不在にすると，そのポートへのすべての I/O 操作で，例外が発生します。その例外をシステムが処理して，あたかも仮想 86 モードタスクが I/O 命令を実行したかのように I/O に出力するか，あるいは I/O からの入力を物理ページに格納したあとにマッピングします。

◻保護付きの割り込み

ハードウェア割り込み

仮想 86 モードでは，割り込みを許可する STI 命令は IOPL と無関係ですが，割り込みを禁止する CLI 命令は IOPL の影響を受けます。このことを

- **STI は仮想 86 モードで IOPL センシティブではない**
- **CLI は仮想 86 モードで IOPL センシティブである**

といいます。安全性の高さを目的としたシステムの場合，IOPL は 3 より小さい値にするため CLI は禁止され，仮想 86 モードからの割り込みは禁止できません。

ソフトウェア割り込み

仮想 86 モードでは，ソフトウェア割り込みについて，8086 と次のような点が異なります。

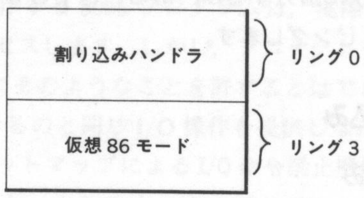
ソフトウェア割り込みでリングレベル 0 に移行

たとえば，仮想 86 モードで MS-DOS が動いているとき，システムコールはリング 0 で実行されます。前に説明したように，リング 0 の割り込みハンドラは，仮想 86 モードを抜けて通常プロテクトモードで動作します。そのため INT*n* で呼ばれるシステムコール実現部は，386 固有のコードです。

仮想 86 モードにおける割り込み/例外

8086 では，ハードウェア割り込みやソフトウェア割り込み，または例外を受け付けると，割り込みベクタテーブルの該当エントリを参照して割り込みハ

● fig.2.25 ●リング3の仮想86モードとリング0の割り込みハンドラ



ンドラを起動します。8086の割り込みベクタテーブルは，物理アドレス0x0から始まる大きさ256×4バイトの表です。1エントリは割り込みハンドラルーチンの開始アドレスを指定するベースとオフセットのそれぞれ16ビットの合計32ビット(4バイト)です。256種類の割り込み/例外を扱えます。それに対し仮想86モードでは，通常プロテクトモードと同様にIDTの該当エントリを参照し，ゲートに従って割り込みハンドラを起動します。

仮想86モードで特徴的なのは，起動される

割り込みハンドラはリング0，通常のプロテクトモードで動作

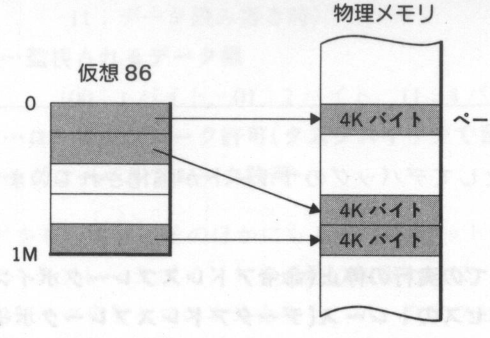
することです。仮想86モードを使うシステムは，システムはリング0，アプリケーションはリング3で動作することを想定しています(fig.2.25)。

また，仮想86モードでIRETが実行されると，例外を発生させます。

回ページング

仮想86モードでは，セグメンテーションが8086のように行われ，セグメントディスクリプタを使用しません。生成される1Mバイトまでのアドレスがリニアアドレスになります(fig.2.26)。ページング機能を活用すると，386の32ビット物理アドレス空間上のページフレームを，自由に仮想86モードのリニアアドレス空間にマッピングできます。セグメントディスクリプタを使用しな

● fig.2.26 ● 1M バイト空間のページマッピング



いので，アクセス制限を受けるのは主にページングレベルの制限です(詳しくは，本章の「4　ページング」p.64 を参照)。

8 デバッグサポート

386では，CPUとしてデバッグのサポートが強化されています。デバッグでは，

- 特定アドレスでの実行の停止（命令アドレスブレークポイント）
- 変数へのアクセスのトレース（データアドレスブレークポイント）

が主なものです。命令アドレスブレークポイントはおなじみでしょう。変数のトレースは，不正なポインタを発見した場合，それがいつ壊されたかを知るために，そのポインタ変数へのアクセスを見つける目的に使われます。しかし，これを純粋にソフトウェアだけで実現すると，とくにデータアドレスブレークポイントを設定したときに実行がたいへん遅くなってしまいます。下手をすると，バグが出る前に日が暮れてしまうほどです。

386では，この2種類のデバッギングを合計4ブレークポイントまで，ハードウェアでサポートします。

8.1 デバッグレジスタ

386にはデバッグハードウェアサポートとして，7個のレジスタがあります。

デバッグアドレスレジスタ…………DR0，DR1，DR2，DR3
デバッグ制御レジスタ………………DR7
デバッグ状態レジスタ………………DR6

デバッグアドレスレジスタにはリニアアドレスが入ります。デバッグ制御レジスタは，DR0からDR3の振舞いを指定し，4つのレジスタそれぞれに応じて，

R/W……… DR0-3 のリニアアドレスでのブレークの方法
（00：命令実行時，01：データ書き込み時，
11：データ読み書き時）
LEN……… 監視されるデータ幅
（00：1バイト，01：2バイト，11：4バイト）
L…………… ローカルブレーク許可（タスクスイッチで自動的に 0）
G…………… グローバルブレーク許可

の4フィールドをもちます。そのほかにも LE，GE ビットがあります。LE，GE ビットはデータブレークポイントを正確に知るために，プロセッサのパイプライン処理でデータアクセスの次の命令を，データアクセス完了まで遅らせます。データアドレスブレークをする際には，これをセットします。

デバッグ状態レジスタは，デバッグ例外が発生したときの発生原因を識別するために使用されます。4つのレジスタそれぞれに対して，

B…………… DR で *n* 例外が発生されると B*n* がセット

ビットがあります。また，

BT……… TSS の T(Trap) ビットによってタスク切り替え時にセット
BS……… EFLAGS の TF(Trap Flag) ビットによってシングルステップ時にセット
BD……… 次の命令がデバッグレジスタにアクセス，ICE を使用時にセット

などのフィールドをもちます。

8.2 デバッギングの例

こうした機能を活用しているデバッガで，デバッグする場合を考えてみましょう。たとえば，グローバル変数 X に書き込んでいるコードを知りたい場合を考えます。デバッガで，ダブルワード変数 X をブレークポイントに設定し

ます。デバッガは，デバッグアドレスレジスタのひとつDR0に，変数Xのリニアアドレスを設定します。さらに，デバッグ制御レジスタのR/W0フィールドに書き込み時ブレークを示す「01」を，LEN0フィールドにダブルワード(4バイト)であることを示す「11」を，DR0によるブレークを有効にするL0ビットに「1」を，正確な場所を知るためにLEビットに「1」をセットします。このようなとき，プログラムが変数Xに書き込みを行うと，DR0レジスタによってデバッグ例外が発生し，例外ハンドラはデバッグ状態レジスタのB0がセットされているのを見て，それを知ることができます。

注意すべきなのは，データアドレスのブレークポイントでは，データのアクセスが終了したあとに例外が報告されるため，データ書き込みの場合は，メモリ内容の変更後に，ブレークポイントが報告されることです。

第3章
MS-DOSと386

1 仮想 86 モードにみる MS-DOS

1.1 秘められた MS-DOS の発展の可能性

386 は，ただ単に高速な 8086 パソコンとしてだけではなく，その優れた機能を利用することによって，本来 8086 用に開発された MS-DOS をさらに発展させる可能性を秘めています。具体的には，

(1)より多くの利用可能なメモリ

(2)複数ある MS-DOS アプリケーションの同時実行

が主なものでしょう。

回メモリ拡張とマルチタスク

とくに，上に挙げた(2)は，一般的に**マルチタスク**と呼ばれているものです。たとえば，長時間かかる計算をするプログラムを走らせたまま，別のプログラムでエディット/コンパイルする，ということが可能です。マルチタスクといってもその実現方式によって形はさまざまですが，現在手にすることのできる環境で比較的ポピュラーなものとしては，MS-Windows 3.0 があります。

MS-Windows 3.0 での MS-DOS アプリケーションのマルチタスクは，CPU が時分割され，実行可能な MS-DOS アプリケーションすべてが CPU スケジュールの対象となって実行されます。上述の例を考えれば，計算をちょっとしてはコンパイル，あるいはエディタをちょっと動かすということを繰り返します。

386 は新たに**仮想 86 モード**という 80286CPU ではもたなかったモードをもち，そのモードでは複数のタスクを管理することができます。その複数あるタスクのひとつひとつを仮想 86 モードで実行することによって，それぞれ独立

した 8086 マシンが複数あるかのようにプログラムを実行できるものです。

複数ある MS-DOS アプリケーションの同時起動についての詳細は，別の機会にゆずるとして，386 が提供する仮想 86 モード&マルチタスク機能による複数 MS-DOS アプリケーションの同時起動は，いわば MS-DOS の横方向への発展ともいうべきものです。作業するための机がいままでひとつしかなかったのに，机を横に並べて同時に作業できる人を増やせるようにした感じです。しかし，ひとつひとつの MS-DOS アプリケーションでできることが，増えたわけではありません。それに対して，ひとつひとつの机を広げることをこの章では考えてみます。ひとつの MS-DOS アプリケーションのできることを増やすのです。具体的な手段として，利用可能なメモリを増やす方法を示します。

1.2 MS-DOS におけるメモリ利用の限界

8086 も最初こそは，8 ビットプロセッサに比べて 16 ビットプロセッサという広大なメモリ空間と処理能力を誇っていましたが，現在ではやや力不足であることは否めません。8086 上で動作する MS-DOS も，8086 の限界の到来とともに，大容量/高速処理という現在の要求に応えられなくなってきました。

しかし，8086 & MS-DOS は圧倒的な普及をみてしまったので，過去の資産をすべて捨て去って次のパソコンアーキテクチャへ移行するのは，ユーザ，ソフトハウス双方にとって相当な負担となります。8086 & MS-DOS の処理能力は別として，まず問題となったのはメモリでした。

すでに説明してきたように，8086 は 1M バイトの物理メモリしかサポートしません。また，MS-DOS ではそのうちの 640K バイトをメインメモリとして使用します。しかも，この 640K バイト程度では，大きなデータを処理する場合には，すぐ足りなくなってしまいます。これは問題です。

2 386によるMS-DOSのためのメモリ拡張

2.1 EMS――8086時代のメモリ拡張方式

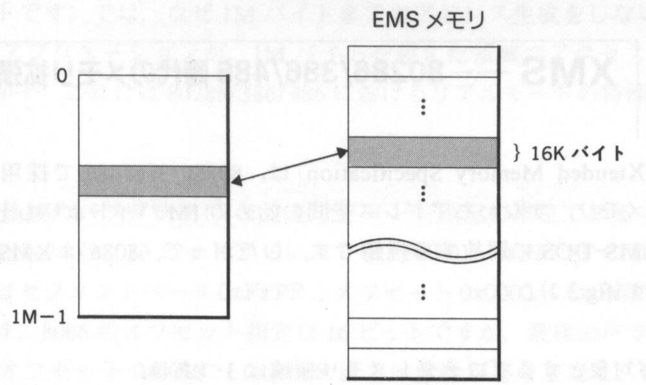
CPUが8086の時代にも，メモリを拡張する方法はいろいろありました。代表的なものとしてはEMS(Expanded Memory Specification)があります。386にも関係のあることなので，ここで簡単にEMSについて説明しておきましょう。

すでにご存知の通り，EMSは，プログラムがEMSメモリを使用する際のインターフェイス規定(メモリそのものの名前ではない)です。この規定では，EMSの割り当てと解放，そして使い方を決めています。逆にいえば，EMSをどう実現するかは，メモリを制御するEMM(Expanded Memory Manager)がインターフェイス規定にそっていれば，ハードウェアであろうがソフトウェアであろうがかまわないわけです。EMSは大まかにいえば，次のような機能を提供します。

- **8086でアクセスできる物理メモリ空間の最初の1Mバイト内にページフレームと呼ばれるメモリ領域を確保する(ページフレームはEMSページで最大64枚分の領域。EMSページ1枚は16Kバイト)**
- **ページフレームに置かれた内容はEMSページ単位で，EMSメモリと呼ばれる別メモリに転送することができる。また，EMSメモリのあるEMSページをページフレームに読み込むことができる**

仮に，EMSメモリを磁気ディスク，ページフレームをメインメモリとみなすと，EMSはディスクを使った仮想メモリシステムといえるでしょう。ページフレームとEMSメモリとの間を，EMSページを単位としたメモリ内容がやりとりされるのです。ページフレームの大きさはEMSページ1枚から指定

● fig.3.1 ● EMS



できますが，あまり少ないと，少ないメインメモリで仮想メモリシステムを構築するようなものです。これは，ページフレームとEMSメモリとの間のやりとりが必然的に多くなり，さらに計算時間に対するデータ転送時間の割合も多くなってしまい，効率上，問題があります。EMSメモリの管理はアプリケーションの責任なので，EMSを利用するアプリケーションは，EMSメモリとページフレームとのやりとりが少なくなるように設計しなければなりません(fig.3.1)。

アプリケーションは，EMS規定にそえば，8086がアクセス可能な1Mバイトの物理メモリの外にあるEMSメモリという物理メモリを利用できます。EMSページを自己管理しなければならない責任という手間は増えますが，最大32Mバイトまでの外部物理メモリを利用できることになります。それまでの1Mバイト以下の制限からみれば，飛躍的な増加といえるでしょう。

EMSは，多くのデータ処理を必要とするLotus『1-2-3』や，MS-Windowsなどで利用されています。EMSに準拠した利用であれば，EMSを利用する複数のアプリケーションが同時に混在していても(たとえば，日本語FEPとワー

プロソフトなど），問題なくEMSメモリを外部メモリ領域として使用できます。

2.2 XMS――80286/386/486時代のメモリ拡張方式

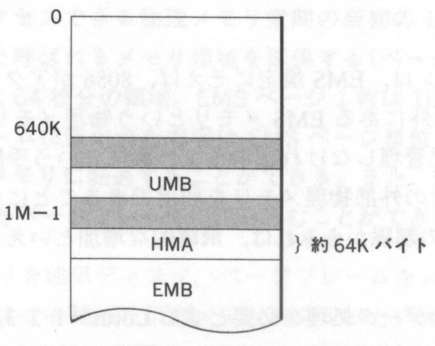
XMS(eXtended Memory Specification)は，80286/386/486で採用されたプロテクトメモリ，すなわちアドレス空間の始めの1Mバイトよりも上位のメモリ空間をMS-DOSに解放する規格です。したがって，8086はXMSの対象外になります(fig.3.2)。

XMSが対象とするプロテクトメモリ領域は3つあり，

HMA(High Memory Area)
EMB(Extended Memory Blocks)
UMB(Upper Memory Blocks)

と呼ばれています。それぞれは，アドレス空間上の位置によって区別されています。

● fig.3.2 ● XMS



回 HMA

この領域は，8086で利用できたアドレス空間の最初の1Mバイトの直後の64Kバイトです。では，なぜ1Mバイトまでのアドレス生成をしない8086用MS-DOSアプリケーションが，1Mバイトを超えた領域にアクセスできるのでしょうか？　これには80286/386/486におけるリアルモードの特殊性が挙げられます。

8086の最後のパラグラフ(メモリをセグメント指定の単位である16バイトで区切ったもの)は，20ビットアドレスで0xFFFF0～0xFFFFFの領域です。この領域はセグメントベース0xFFFFとオフセット0x0000～0x000Fでアクセスされます。8086のオフセット指定は16ビットですが，最後のパラグラフに対して，オフセット0x0010～0xFFFFまでは意味がありません。8086では0xFFFFFを超えるアドレスを生成しても，CPUとして0x00000～0xFFFFF(1Mバイト)までの物理アドレス生成の機能しかもたないためです。そのため8086では，最後のパラグラフについてオフセット0x0010～0xFFFFを指定すると，アドレス空間をぐるっと回って，最初のパラグラフへのアクセスになります。

ベース (16ビット)	：オフセット (16ビット)	＝ 物理アドレス (20ビット)
FFFF	: 000F	= FFFFF
FFFF	: 0010	= 00000

80286/386/486のリアルモードはほとんど8086と同じですが，若干の違いがあります。その違いのひとつが，この「ぐるっと回って最初のパラグラフ」です。

8086では20ビット分のアドレス計算しか行わないため，最後のパラグラフへのオフセット0x0010～0xFFFFの指定は，20ビットからの桁あふれが無視され，0x00000～0x0FFEFへのアクセスとなりました。一方，80286/386/486 CPUでは，1Mバイト以上の物理メモリをサポートしています。そのため，リアルモードで最後のパラグラフへオフセット0x0010～0xFFFFを指定しても20ビットの桁あふれを起こさず，0x100000～0x10FFEFというプロテクトメモリ中のアドレスを生成します。リアルモードから，8086のアドレス指定

である16ビットベース&16ビットオフセットをそのままに，プロテクトメモリがアクセスできるのです。

ベース (16ビット)	:オフセット (16ビット)	= 物理アドレス (20/32ビット)
FFFF	:000F	= FFFFF
FFFF	:0010	= 100000
	⁝	
FFFF	:FFFF	= 10FFEF

このアドレス0x100000～0x10FFEFまでの，(64K－16)バイトプロテクトメモリ領域をHMAと呼び，XMSではMS-DOSアプリケーション用に利用方法を規定しています。XMSにそってHMAを利用するソフトとして，最近ではMS-DOS Ver.5.0があります。Ver.5.0では，MS-DOSの一部をHMAに置くことにより，アプリケーションの利用可能なメモリがアドレス空間の始めの640Kバイト内に(あまり多くはありませんが)，約64Kバイトほど増えています。

□ EMB

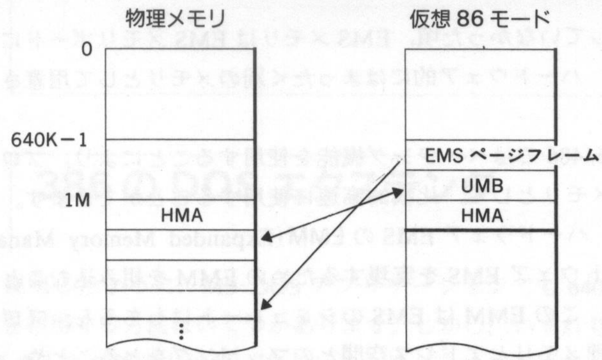
HMAよりも上位にあるプロテクトメモリ領域です。具体的にはアドレス0x10FFF0以上のメモリ領域です。プロテクトメモリ中，HMAまでにはリアルモードでアクセス可能ですが，EMBにはアクセスできません。EMBはEMSメモリのように利用されます。すなわち，EMB領域内のデータは，いったんリアルモードでアクセスできるアドレス空間の最初の1Mバイト内領域へ転送されるわけです。

□ UMB

UMBはアドレス空間の後半，VRAMやBIOS ROMなどが置かれている640K～1Mバイト領域のうち，MS-DOSに利用されていない，「すきま」領域を使います。XMSではUMBの要求/解放を規定します。

さて，このあとのソフトウェアEMSにも関わる，メモリの実際的な実現方

● fig.3.3 ● HMA, UMB, 仮想 EMS



式にまつわる話になりますが，386/486のUMBのRAMは，プロテクトメモリ領域のRAMが使われます。640Kバイト以上には，MS-DOSが自由に使用するためのRAMが実装されていないので，386/486のページング機能によって，プロテクトメモリをリアルモードでアクセスできるUMB領域にマッピングします。それをMS-DOSアプリケーションでアクセスできるようにして，UMBを利用します。この場合，ページング機能を有効にするため，MS-DOSアプリケーションは仮想86モードで動作させます(fig.3.3)。

以上が，XMSで操作規定されている拡張メモリです。XMSに従っているアプリケーションが混在しても，HMA，EMB，UMBは正しく使われます。

2.3 ソフトウェアEMS——386/486のページング機能を使用

386が普及していなかった頃，EMSメモリはEMSメモリボードによって実現されており，ハードウェア的にはまったく別のメモリとして用意されていました。

しかし，386/486ではページング機能を使用することにより，プロテクトメモリをEMSメモリとして，比較的高速に使用することができます。ソフトウェアEMSは，ハードウェアEMSのEMM(Expanded Memory Manager)とは異なる，ソフトウェアEMSを実現するためのEMMを組み込むことによって実現されます。このEMMはEMSのシミュレートはもちろん，プロテクトメモリなどの物理メモリとアドレス空間とのマッピングをとることや，物理メモリの管理などを行います。

XMSで定めたUMBも，このEMMの機能の一部によって提供されます。

ソフトウェアEMSでは，プロテクトメモリがEMSメモリとして利用されます。ページフレームのEMSページ1枚には，386/486のページング機能によりプロテクトメモリの物理ページ4枚(16Kバイト)がマッピングされます。ページテーブルエントリの更新だけでページフレームへのEMSページの取り込みができるので，16Kバイトのデータ転送が必要ありません。CPUが十分高速で，またメモリアクセスも十分高速な場合，EMSをシミュレートしているオーバーヘッドを考慮しても，ハードウェアEMSと遜色のない使い勝手が得られます。ソフトウェアEMSでは，EMSページへのアクセスは内部バスによるアクセスだけですみますが，EMSメモリボードによる場合は外部バスを通じてのアクセスが入ります。CPUがますます高速になる現在では，この点でもソフトウェアEMSには将来性があると思われます。

3 DOSエクステンダの登場

3.1 386のDOSエクステンダ

前節で説明したように，MS-DOSアプリケーションでも640Kバイト以上のメモリを利用する方法はいくつかあります。しかし，いずれも1Mバイトという8086の小さな領域と拡張メモリとをやりとりする方法で，管理に手間がかかります。しかも，データ転送のための時間がかかるのも難点です。

そこで，プロテクトモード本来の機能を生かすべく登場したのが**DOSエクステンダ**です。DOSエクステンダには80286用もありますが，ここでは主に386用のものについて説明します。

386/486のプロテクトモードでは，4Gバイトまでの物理アドレス空間に対してプログラムから容易にアクセスが可能です。EMSメモリにみるEMSページとのやりとりなどは行わなくてもすむわけです。DOSエクステンダの特徴のひとつには，スモールモデルしかないことがあげられます。386では1セグメントが最大4Gバイトであり，これは全アドレス空間に匹敵します(スモールという呼び方は，やや的外れですね)。そのため，コードやデータを分割して小さなセグメントにつめ込むということをしなくてすみます。また，アドレスの指定が事実上オフセット値のみによって行われることから，UNIXで養われた技術の移植が，よりスムーズになりました。UNIXで多くの人から利用されている評判のツールで，MS-DOS用に移植されたものには，現在，「gcc」,「NEmacs」,「JTeX」などがあります。86系CPUも486の時代になり，一頃のRISCと変わりない性能になっているので，こうしたツールを身近に利用できるようになったことは，会社や大学でUNIX漬けになっている方に福音ではないでしょうか。

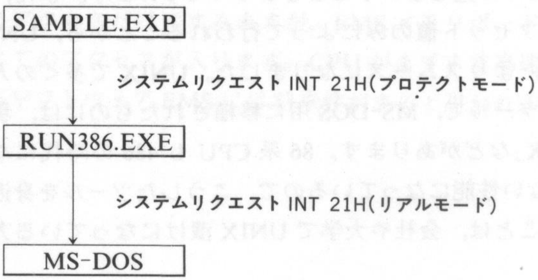
さて，DOSエクステンダの目標は，MS-DOSを基盤としてプロテクトモードで走るアプリケーションを実現することです。DOSエクステンダ環境で実行されるアプリケーションは，大まかにいえば，MS-DOS用のコードとプロテクトモード用のコード，そしてDOSエクステンダ自身から構成されます。典型的なDOSエクステンダ用プログラム（拡張子が〝.EXP〟）を引数にして，DOSエクステンダ起動プログラム（『RUN386』――Phar Lap社のDOSエクステンダ――など）を実行します。たとえば，

```
A>RUN386 SAMPLE.EXP 1024 ↵
```

のようにします。この例では，〝SAMPLE.EXP〟というDOSエクステンダ用プロテクトモードプログラムを，〝RUN386.EXE〟とともに起動しています。1024は〝SAMPLE.EXP〟への引数です。DOSエクステンダは，ほかにもフリーウェアのものもありますので，「gcc」とともにMS-DOSで手軽に386/486固有の世界を楽しめます。

ここでは『RUN386』を参考に，引き続き，この例を使った動作を説明します。

プロテクトモードプログラムを，わざわざMS-DOSから起動することの意味に疑問をもつ人がいるかもしれません。MS-DOS上で走ることの利点は，ファイルシステムや画面表示などのシステムルーチンを流用できることです。そのためプロテクトモードから発行する，各種サービスコールをDOSエクステンダはMS-DOSに処理させ，結果をまたプロテクトモードのプログラムへ渡すという仲立ちをします。



ここで注意しなければならないことは，MS-DOSがリアルモードで動作することです。このためDOSエクステンダは，プロテクトモードプログラムが発行したシステムコールINT 21Hを受け取ると，リアルモードに自ら切り替え，リアルモードの環境設定をしたあとに，MS-DOSにシステムコールを発行し直します。システムサービスの要請を受けたMS-DOSから見れば，システムコールを発行したのはDOSエクステンダなわけです。したがって，MS-DOSのシステムサービスが終了してIRETすると，いったんDOSエクステンダに戻ります。そして，DOSエクステンダはIRETしてプロテクトモードプログラムに戻ります。

おもしろいのは，DOSエクステンダがシステムコールの仲立ちをする際に，システムコールの翻訳もすることです。システムコールの中には，データ転送先の領域をポインタで指定するものがあります。しかし，プロテクトモードから発行されるシステムコールでプロテクトメモリ領域をデータ転送先として指定しても，リアルモードであるMS-DOSからはアクセスできません。DOSエクステンダはこうした場合に備えて，DOSからアクセス可能な1Mバイトの領域内にバッファをもちます。そして，MS-DOSへのシステムコールのデータ転送先をMS-DOSがアクセス可能な，そのバッファに指定します。MS-DOSへのシステムコールが終了したあと，内部バッファからプロテクトメモリへ内容を送ります。

こうしたDOSエクステンダの内部バッファを介したデータ転送の際，データ転送量が内部バッファのサイズよりも大きい場合には，システムコールを複数回に分けて発行します。たとえば，DOSエクステンダの内部バッファのサイズが1Kバイトしかないときに，〝SAMPLE.EXP〟が4Kバイトのバッファを必要とするシステムコールを発行すると，DOSエクステンダはそのシステムコールを4回に分けて発行し，毎回のシステムコールのあとに，内部バッファからプロテクトメモリ上のバッファにデータを再転送します。



DOSエクステンダ自身は一種のシステムコールライブラリで，INT 21Hを独自解釈します。先ほどまで説明してきたMS-DOSのシステムコールも，ファンクション2511Hというシステムコールで，各種レジスタにMS-DOSシステムコール用の値を設定することで実現されています。そのほか，プロテクトモードプログラムがリアルモードルーチンと協調して実行するのに必要なサービスルーチンを用意しています。

さて，DOSエクステンダをMS-DOSのプロテクトメモリの使用という視点からみると，MS-DOSのほかのプロテクトメモリの使用方法と衝突する可能性があります。たとえば，ソフトウェアEMSでは386/486の仮想86モードでのページング機能を利用して，プロテクトメモリをMS-DOSがアクセス可能な1Mバイトの領域にマッピングし，EMSページとして使用していました。しかし，DOSエクステンダもプロテクトメモリを使用します。このためソフトウェアEMSを実現するEMMと，DOSエクステンダが協調して動作しないと，ソフトウェアEMSを使用しているMS-DOSからDOSエクステンダを使うとシステムが破綻してしまいます。そこで決められたのが，次に説明するVCPIです。

3.2 VCPI——ソフトウェア EMS と DOS エクステンダ共存のための規定

VCPI(Virtual Control Program Interface)は，1989 年の 6 月に決まりました。VCPI 規格では，386 の MS-DOS 上で走るコントロールプログラム(OS とまではいかないが MS-DOS を補うプログラム)を，次の 3 つに分類しています。

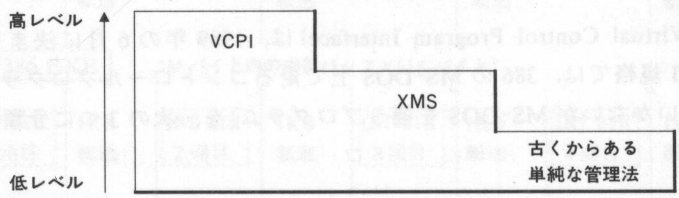
(1)**プロテクトモードのランタイム環境**(例：DOS エクステンダ)
(2)**EMS エミュレータ**(例：ソフトウェア EMS 用 EMM)
(3)**マルチタスク環境**(例：MS-Windows/386)

(1)の環境にあるプログラムはプロテクトモードで走ります。(2)と(3)は，仮想 86 モードで実行されます。こうしたコントロールプログラムを MS-DOS 上で同時に使いたい場合がありますが，互換性がないと，ひとつを走らせるときはほかをやめなければいけないということが生じます。そこで，DOS エクステンダ/EMS エミュレータ/マルチタスク環境の 3 種類のコントロールプログラムが同時に問題なく走れるようにインターフェイスを決めたものが，VCPI 規格というわけです。

こうしたコントロールプログラムの混在の際に問題を起こすのは，仮想 86 モードを使った EMS エミュレータです。EMS エミュレータは，386 が登場するとすぐに広く普及したので，ほかのコントロールプログラムとの互換性をはかるために VCPI は，EMS Ver.4.0 インターフェイスのスーパーセットとして決められました。互換性のある EMS ドライバは，VCPI の元につくることになります。典型的な使用法としては，VCPI を提供する EMS Ver.4.0 互換の EMM を使用します。また，こうした EMM は VCPI を提供するため，VCPI **サーバ**と呼ばれます。

VCPI には，大きく分けて仮想 86 モードプログラム用のインターフェイスと，プロテクトモードプログラム用のインターフェイスがあります。仮想 86 モードプログラムの VCPI は INT 67H，プロテクトモードプログラムの VCPI

● fig.3.4 ● VCPIのレベル



はFAR CALLで規定されています。たとえば，仮想86プログラムが物理メモリを確保するためにはINT 67HファンクションDE04H，プロテクトモードプログラムが物理メモリを確保するためにはFAR CALLファンクションDE04Hなどが決められています。DEHはVCPI準拠のEMS用ファンクションコードで，04HがVCPIファンクションコードです。INT 67HはEMSインタラプトです。VCPI規格にそったプログラムがVCPIを使用できるときには，物理メモリを確保する際に必ずこのインターフェイスを使って行わなければならないわけです。このVCPIの実現部分（VCPIサーバと呼ぶ）が，複数のコントロールプログラムが互いに不正な操作をしないことを保証します。

さて，インターフェイスを規定すると互換性のあるプログラムがつくれることについて，もう少し説明を続けましょう。共存によって生じる問題にはいろいろありますが，いま注目しているのはメモリの問題なので，メモリの場合を例に説明します。

プロテクトメモリの管理には，まず従来から行われていた低レベルな管理方法があります。これはプロテクトメモリをひとつの連続領域として管理する方式で，融通のきかない原始的な方式です。そこでより高級な管理方法としてXMSという規格が考えられました。そのXMSがHMA，EMB，UMBの3つの領域を管理することはすでに説明しました。そして，VCPI規格があるわけです。VCPI準拠プログラムとしては，プロテクトメモリの使い方を，次の優先順位で行います（fig.3.4）。

① VCPI
② XMS
③ 古くからある単純な管理法

VCPIに準拠しているコントロールプログラムは，VCPIで統一されているパソコン環境では問題なく動きます。逆にVCPIに準拠していなければ，そのプログラムはほかのVCPI準拠コントロールプログラムと問題なく混在できることは保証されません。Phar Lap社のDOSエクステンダはVCPIに準拠しています。初期のEMSエミュレータはVCPIに準拠していませんので，注意が必要です。

3.3 DPMI――さらに大きな枠組み

DPMI(DOS Protected Mode Interface)は，XMSやVCPIと同様に，プロテクトメモリをMS-DOSから利用するために1990年5月に発表された規約です(Ver.0.9)。現在のバージョンは，1.0です。VCPIが主にEMSエミュレータとDOSエクステンダのメモリ使用の共存を狙っていたのに対し，DPMIではもっと広く，MS-DOS上でプロテクトモードプログラムを動作させるコントロールプログラムのインターフェイスの一部まで規定しています。したがって，DPMI準拠とうたっている環境では，ひとつのMS-DOSセッションで，やはりDPMI準拠のプロテクトモードプログラムを問題なく動作できることになります。VCPIと同様に，DPMI準拠のDOSエクステンダは，DPMI準拠のEMSエミュレータと混在できます。

VCPIのときと同様，DPMI準拠プログラムとしては，プロテクトメモリの使い方として，次の優先順位で行います。

① DPMI
② VCPI
③ XMS
④ 古くからある単純な管理法

DPMIが提供されている(DPMIホストが存在している)のに，DPMIを使用せずにプロテクトメモリを使用するプログラム(DPMI非準拠プログラム)が実行されると，(DPMIの目的からすれば)拒否されるでしょう。

DPMI Ver.1.0 —— INT 31H DPMIサービス一覧

DPMIが提供するサービスの主なものには，次のようなものがあります(*印は，DPMI Ver.0.9に付加されたVer.1.0のサービス)。

● LDT管理サービス

LDTディスクリプタの割り当て/解放
リアルモードセグメントのLDTディスクリプタへのマップ
セレクタ増分値の取得
セグメントベースアドレスの取得/設定
セグメントリミットの設定
LDTディスクリプタアクセス権の設定
エイリアスLDTディスクリプタの作成
LDTディスクリプタの取得/設定
指定LDTディスクリプタの割り当て
*複数LDTディスクリプタの取得/設定

●拡張メモリ管理サービス

フリーメモリ情報の取得
メモリブロックの割り当て/解放/サイズ変更
*メモリブロックのサイズとベースの取得
*ページアラインメモリブロックの割り当て/サイズ変更
*ページ属性の取得/解放
*デバイス/コンベンショナルメモリのメモリブロックへのマップ
*メモリ情報の取得
物理アドレスマッピング

＊物理アドレスマッピングの解放
＊共有メモリの割り当て/解放
＊共有メモリのアクセスシリアライズ/シリアライズ解放

● MS-DOS メモリ管理サービス

MS-DOS メモリブロックの割り当て/解放/サイズ変更

●割り込み管理サービス

リアルモード割り当てベクタの取得/設定
プロセッサ例外ハンドラベクタの取得/設定
プロテクトモード割り込みハンドラベクタの取得/設定
＊プロテクトモードの拡張プロセッサ例外ハンドラベクタの取得/設定
＊リアルモードの拡張プロセッサ例外ハンドラベクタの取得/設定
仮想割り込み状態の取得と無効化/有効化
仮想割り込み状態の取得

●変換サービス

リアルモード割り込みのシミュレート
FAR リターンフレームでのリアルモードプロシージャのコール
割り込みリターンフレームでのリアルモードプロシージャのコール
リアルモードコールバックアドレスの割り当て/解放
ステート保存/復帰アドレスの取得
ロウ CPU モードスイッチアドレスの取得

●ページ管理サービス

リニア領域のロック/アンロック
リアルモード領域のページ可能へのマーク
リアルモード領域の再ロック
ページサイズの取得
ページのデマンドページング候補としてのマーク
ページ内容の無効化

● **デバッグサポートサービス**

デバッグウォッチポイントの設定/解除/リセット
デバッグウォッチポイントの状態の取得

● **そのほかのサービス**

DPMIバージョンの取得
* DPMIケーパビリティの取得
ベンダー固有のAPIエントリポイントの取得
*常駐サービス提供コールバックのインストール
*終了して常駐
*コプロセッサ状態の取得
*コプロセッサエミュレーションの設定

「DOS PROTECTED MODE INTERFACE（DPMI）SPECIFICATION」Version 1.0より

DPMIの目的は，VCPIがDOSエクステンダアプリケーションのマルチタスクに向かなかった点を補うことであり，DPMIとVCPIがもっとも違う点は，DPMIではDPMIサーバがアプリケーションよりも高い特権レベルで中央集権的に動くことです。VCPIは，VCPIクライアントがリング0で実行することを許可してしまうので，アプリケーションが使うインターフェイスとしては向きません。

3.4 DPMI Ver.1.0とDOSエクステンダの関係

前述したように，DPMIは，マルチタスクのMS-DOS環境でDOSエクステンダなどによるプロテクトモードアプリケーションが，正常に動作するための規格です。この規格を遵守してDPMIのファンクションコールを行うプログラムを，**DPMIクライアント**といいます。それに対して，そのファンクションコールを処理する側を**DPMIホスト**といいます。現在，DPMIホストの典型的な例は，MS-Windows 3.0（以下，Windows 3.0と略）です。

現在，DPMIのバージョンには，0.9（Windows 3.0が対応）と，その上位の

1.0とがあります。近いうちに，Windowsの次のバージョンや，OS/2 Ver.2.xのDOS互換ボックスでは，Ver.1.0に対応することでしょう。本書では，DPMI Ver.1.0をもとに解説していきます。Ver.0.9の問題点については，この章の最後にまとめてありますので，そちらを参照してください。

では，DPMIとDOSエクステンダの関係について解説していきます。DPMI Ver.1.0におけるDOSエクステンダのプログラミング上の役割をつかんでください。

DPMIホストが存在する環境では，DPMI対応のDOSエクステンダアプリケーションはDPMIクライアントとして動作します。逆にDPMI対応ではない，たとえばVCPIにしか対応していないDOSエクステンダアプリケーションは，特権レベルによる保護によってDPMIホストから実行を拒否されるでしょう。これは，VCPIまでの対応では，DOSエクステンダアプリケーションが特権モードで実行しようとするからです。その理由は，VCPIサーバと，いくつかのDOSエクステンダ機能が行わなければならない作業は，特権レベル0でしか操作できない性質のもので，システム全体にかかわることだからです。しかし，そうした特権レベル0で行わなければならない作業は，その作業を必要とするプログラムに共通です。したがって，そうしたシステムに重要なリソースを操作する「危険な」作業を，DPMIホストに一元化して行うことによって信頼性が増し，システム全体は安全に動作します。これがDPMIの意義です。

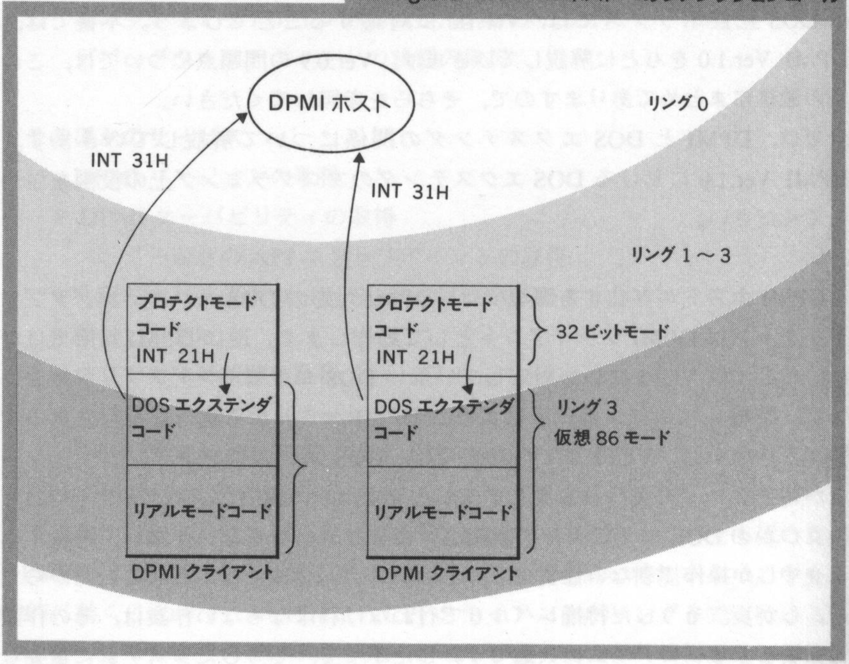
このDPMIホストは，DOSエクステンダを含むプロテクトモードからの拡張システムコールを受け付けます。例外は，INT 2FHの次のファンクションで，リアルモードからも受け付けます。

●リアルモードからプロテクトモードへの切り替えエントリポイントの獲得
（INT 2FHファンクション1687H）

これはDPMIホストが存在していれば成功し，

DPMIホストの存在確認

● fig.3.5 ● DPMIホストへのファンクションコール



のために使います。また，プロテクトモードの「最初に」切り替えるためのプロシージャ「セグメント：オフセット」を返します。これ以外のDPMIファンクションコールはプロテクトモードから発行され，INT 31H(一部INT 2FH)によって行われます(fig.3.5。これ以降の説明で，DPMIのファンクションコールを指すとき，とくに断りがないときはINT 31Hだと考えてください)。

たとえば，DPMI対応のDOSエクステンダとして386|DOS-Extender Ver.4.1があります。この製品では，プロテクトモードコードはDOSエクステンダのサービスを受けるために，拡張ファンクションコールをINT 21Hで呼び出します。INT 21Hで386|DOS-Extenderに入ったあとは，さらにINT 31HでDPMIファンクションコールを行い，システムに対してクリティカルな操作をDPMIホストにゆだねます。

□ DPMIクライアントの動作

ここで，DOSエクステンダプログラムに代表されるDPMIクライアントが，一般的にどのような動作をして，マルチタスク環境下でのDOSプロテクトモードプログラムを正常に動作させているかについて注目してみましょう。これは，将来のWindows 3.xやOS/2 Ver.2.x，さらにはUNIXのDOSボックスでDPMIクライアントが動く場合など，すべてに当てはまるものです。

クライアント初期化

DPMIクライアントは，最初，DOSからリアルモード(仮想86モード)で起動します。その後，次の手順でプロテクトモードに切り替わります。

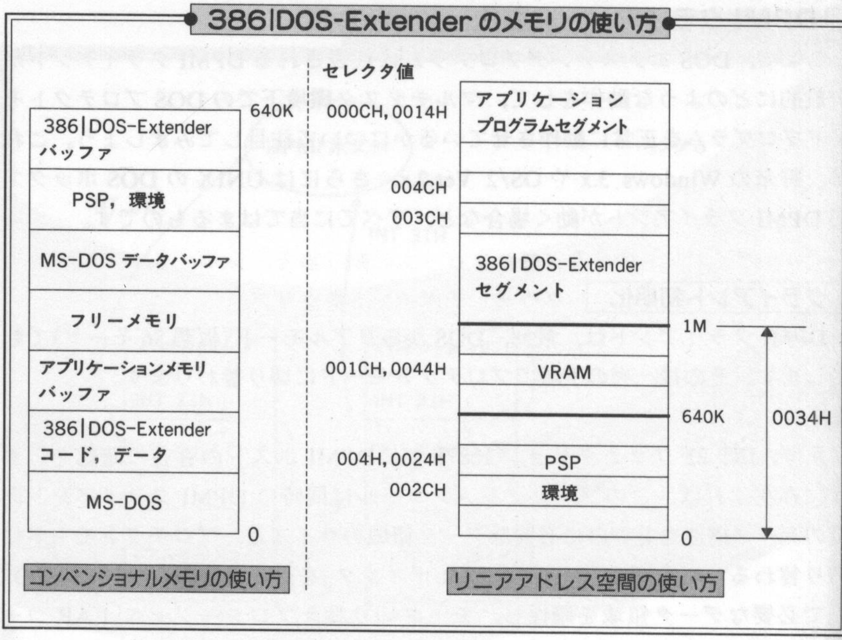
まず，INT 2Fファンクション1687Hで，DPMIホストの存在の確認をします。存在すれば，このファンクションコールは同時にDPMIクライアント固有のデータ構造をもつのに必要なデータ領域のサイズと，プロテクトモードに切り替わるためのプロシージャ「FARポインタ」を返します。引き続き，DOS上で必要なデータ領域を確保し，モード切り替えプロシージャへ「FARコール」します。

無事プロテクトモードに切り替わると，CS，SS，DSの各セグメントレジスタには，プロテクトモードに切り替わる前の，各対応リアルモードセグメント(CS，SS，DS)にアクセスするためのセレクタが入っています。つまり，プロテクトモードになっても内容的には，リアルモードと同じ「64Kバイト」セグメントを引き続き使いながら走っています。一方ESは，プログラムのPSPへのセレクタをもっています。

クライアント終了

DPMIクライアントは，DOSでリアルモードプログラムがINT 21H AH=4CHで終了するのと同じスタイルをとり，プロテクトモードでも同様に，INT 21H AH=4CHで終了します。つまり，DPMIホストが提供するプロテクトモードプログラム用のINT 21H用のデフォルトのハンドラは，AH=4CHでクライアントを終了させるようになっています。

終了するにあたって，クライアントが事前にしておかなければならないのは，



次のようなことです(以下の説明で，「メモリをロックする」とは，指定したメモリを含むページを物理メモリに置いたままにすることを意味し，「アンロックする」とは，その逆で，仮想記憶のためにディスクへ退避する対象としてページを指定することを意味します)。

- **ページングのためにアンロック**(INT 31H AX=0603H)**したリアルモードメモリをロック**(DPMI Ver.0.9)
- **リアルモード用フック**(INT 31H AX=0201H)**の割り込みハンドラを復帰**
- **リアルモード例外のためのプロテクトモードハンドラ**(INT 31H AX=0213H)**を可能ならクリア**

INT 21H ファンクション 4CH を受けた DPMI ホストが行う作業は，次の通りです。

- **割り当てられた拡張メモリをアンロック/解放**(DPMI Ver.0.9)

- クライアントの LDT を解放(DPMI Ver.1.0)
- 物理アドレスマッピングを解放
- デバイスやコンベンショナルメモリマッピングを破棄
- クライアントの IDT を解放(DPMI Ver.1.0)
- アンロックされたリアルモード領域をロック(DPMI Ver.1.0)
- 割り当てられたリアルモードコールバックルーチンを破棄
- クライアントの共有メモリブロックの割り当て，シリアライズを解放
- セットされたデバッグウォッチポイントをクリア
- コプロセッサ状態をデフォルトに復元

DPMI ホストは，その後リアルモードに切り替わり，DOS へ INT 21H ファンクション 4CH をコールして，リアルモードの DPMI クライアントの終了を DOS にゆだねます。

割り込みの処理

DPMI クライアントがプロテクトモードに入ると，専用の IDT が割り当てられます。DPMI Ver.0.9 では，全体でひとつの IDT を参照します。それに対して，DPMI Ver.1.0 では，クライアントごとに別々の IDT が提供されます。IDT の中身は，

- INT 31H　DPMI ファンクション
- INT 2FH　DPMI ファンクション
- INT 21H AH=4CH　クライアント終了

がプロテクトモードで処理されるように設定されているほかは，リアルモードで割り込み処理するようになっています。上に示した以外のソフトウェア/ハードウェア割り込みは，デフォルトハンドラによってリアルモードでの割り込みに変換され，DOS の割り込みハンドラで処理されます。クライアントは，プロテクトモード/リアルモードの，独自の割り込みハンドラを設定することもできます。

第 2 章の 386 アーキテクチャの説明で述べているように，マルチタスク化さ

れているプログラムの割り込みフラグ(IF)操作によって，割り込みを禁止することは自由にできません。それを許してしまっては，システムが安全ではなくなるからです。しかし，DOSプログラムにはシングルタスクの条件でIF操作しているものがあります。そのようなDOSプログラムを「マルチDOS環境(DOSを複数起動できる環境)では動かない」としてしまうのは簡単ですが，これではあまりにも不親切です。そこで，マルチDOS環境に対応するために，IFの操作をする命令実行が拒否されて例外が生じると，DPMIホストは例外を処理する段階で，

仮想割り込みフラグ

に対する処理で置き換えます。したがって，DOSプログラムのIF操作命令は，CPUのIFには何の影響ももちません。その上，このプログラム実行中のハードウェア割り込みは受け付けられず，見かけ上，IF操作命令によって割り込みが禁止されたことと同じ効果を得ています。仮想割り込みフラグは，DPMIファンクションによっても変更されます。

共有メモリの使用

共有メモリは，DPMI Ver.1.0の機能です。一般的に共有メモリとは，並列に動作する複数のプログラムが内容を共有でき，通信などに使う仕掛けです。このあとで説明するWindowsのクリップボードも，意味的には共有メモリといえます。

DPMIが提供する共有メモリは，仮想マシンの32ビットアドレス空間の一部を，別の仮想マシンの32ビットアドレス空間の同じ場所と共有できる機能です。

本来，アドレス空間は仮想マシン(タスク)ごとに固有で独立したものです。しかし，それぞれの仮想マシンの，あるセグメントについて，

- **セグメントディスクリプタで指定されるセグメントベースが同じリニアアドレスをもち，**
- **しかも，ページングでそのセグメントが同じ物理ページにマッピングされる**

ならば，異なる仮想マシンで同じ物理ページを参照できます。

「メモリ」という文字通りに，同じリニアアドレスをもつメモリ領域を共有して使います。ただ共有するだけなら，同じリニアアドレスで複数の仮想マシンに見える必要はないのですが，アドレスを格納する場合には同じほうが都合がよいのです。

これをもっとわかりやすくいうと，たとえばプログラム A が共有メモリ中の最初の 1 バイトに文字「a」を書くと，その共有メモリを共有するプログラム B が，最初の 1 バイトを見たとき「a」と見えるものです。

共有メモリを得るには，次のような手順を踏みます。

共有メモリブロックに名前，サイズを指定して割り当てる(INT 31H AX=0D00H)。ハンドルや先頭リニアアドレスなどが返される

↓

共有メモリブロックに LDT セレクタを割り当てる(各種INT 31H)。セレクタが使えるようになると，プログラムからアクセスできる

共有メモリには，各プログラムがまちまちにアクセスするのではなく，多くの場合，一時にひとつのプログラムだけがアクセスできるようにする，などのアクセス制限があります。このことを**シリアライズ**といいます。このシリアライズのインターフェイスは，DPMI で規定されています。

また，シリアライズにもとづいた共有メモリのアクセス要求については，次のようになっています。

ハンドルを指定してアクセス要求する(INT 31H AX=0D02H)。
要求タイプには，排他アクセスか共有アクセスか，また要求が満たされるまで待つか，満たされなかった場合は終了するかのどちらかを指定

排他アクセスを要求すると，指定した共有メモリが，すでに共有/排他アクセスでほかのプログラムに確保されている場合，要求はブロックされます(要求は満たされません)。また，共有アクセスを要求すると，排他アクセスでほかのプログラムに確保されている場合に，要求はブロックされます。これらのアクセスモードは，

共有メモリブロックの内容を変更したい　→　排他アクセス
共有メモリブロックの内容を読み出したい　→　共有アクセス

のように使われます。

こうしてシリアライズによって共有メモリブロックへのアクセス権を得たあとにアクセスを行います。アクセスが終了し，アクセス権を放棄するには，次のようにします。

ハンドルと，放棄するアクセス権のアクセスモード（共有/排他）を指定して，アクセス権を解放

その際，解放しようとしているアクセス権でブロックされたアクセス権要求を，強制解除するかどうかを指定できるようになっていますので，指定があればそれを実行します。用途としては，たとえばアクセス権要求でブロックされたプログラムに割り込みをかけて，そのプログラム（仮想マシン）の割り込みハンドラが，そのプログラムのブロック原因のアクセス権要求を強制解除することが考えられます。

4 MS-DOS と C 言語

4.1 8086 上での C 言語

MS-DOS は 8086 をベースとした OS のため，MS-DOS 上で大きなデータやコードを扱うプログラムの場合は，64K バイトのセグメントをとっかえひっかえして，メモリにアクセスしなければなりません。

こうした 8086 のアーキテクチャ上の特徴から，MS-DOS の C 言語プログラミングでは，メモリモデルというものが存在しています。各メモリモデルで，コードセグメント，データセグメント，スタックセグメントに単一，あるいは複数のセグメントを使うことを許すかどうかは，次の通りです。

	コード	データ	スタック
スモールモデル	1	1（データ＋スタック）	
ミディアムモデル	複数	1（データ＋スタック）	
コンパクトモデル	1	複数	1
ラージモデル	複数	複数	1（静的データ＝1 セグメント）
ヒュージモデル	複数	複数	1（静的データ＝複数セグメント）

＊メモリモデルの呼称は，Microsoft 社のもの

スモールモデルのように，データセグメントがひとつのセグメントのみによって実現される場合は，データをアクセスするためのオフセットは 16 ビットですみます。たとえば，

```
main() {
    char *p, *c;

    c = malloc(100);
    …
}
```

というCプログラムの場合，スモールモデルでコンパイルできるので，ポインタ「p」は16ビットです。データが1セグメント内に納まる場合，C言語のポインタは16ビットになります。これがNEARポインタです。8086は16ビットCPUなので，

```
*p++ = *c++;
```

のようなアドレス計算を含むステートメントに対しては，レジスタを使ったオフセットのみによるアドレス計算によってコードの最適化を行い，速いコードを生成できます。

一方，ラージモデルのようにデータセグメントが複数のセグメントによって実現される場合には，ポインタを16ビットで表現することはできません。よって，アクセスする物理アドレスを生成するために，セグメントベースとオフセットを指定しなければなりません。そのために，それぞれ16ビットずつ，合計32ビットが必要です。FAR，HUGEポインタがこれに当たります。とくに，HUGEポインタは，ポインタのインクリメントの際，セグメント境界をまたぐときにセグメントベースの更新を行うので，64Kバイト以上のサイズをもつ領域のポインタ移動でセグメントを意識しなくてすみ，使い勝手はよくなっています。

このように8086でCプログラミングをするときには，C言語の特徴のひとつであるポインタに，16ビットと32ビットの2通りが存在することになります。しかも，このことは「セグメント境界をまたぐかどうかの監視」という余分な作業を必要とするので，パフォーマンス上問題です。セグメントは，あくま

でコードやデータといった「種類の違い」のためだけにあるべきであって，連続領域として指定できるサイズの制限は余分でした。

4.2 386 プロテクトモードと C 言語

本来，C 言語は MS-DOS 上より，UNIX のもとでその真価を発揮します。UNIX は，主にワークステーション上で動作しますから，そこで採用しているほとんどの CPU では，アドレッシングにリニアアドレスが使用されます。32 ビット CPU であれば 32 ビットのポインタしか存在しないので，プログラミングや生成されるコードはとてもすっきりしています。同様に，386 パソコンのプロテクトモードでも，そうしたワークステーションの CPU と同等のことが可能です。

386 では，1 セグメントのサイズが最大 4G バイト，つまり 32 ビットリニアアドレスすべての領域に指定可能です。そのため，大きなコードやデータでも，常にオフセット＝ポインタとすることができ，先に述べたワークステーションの CPU と同じ環境になります。386 の「最大セグメント＝アドレス空間」は，そうしたことを実現することを目的としたものでもあるわけです。ひとつの種類の領域が複数のセグメントにまたがることによるセグメントレジスタ（セレクタ）の管理が不用になり，ポインタ＝32 ビットオフセットとみなして，プログラミングまたはコード生成ができます。そのため UNIX ワークステーション上で動作している，C で書かれた多くのアプリケーションが，楽に 386 へ移植できます。もっとも移植に限ってみれば，「NEAR，FAR の問題」というよりは，「8086 が最大でも 1M バイトの物理メモリしか扱えない」ということが大きな問題になります。なぜなら，8086 で数 M バイト単位のメモリを扱おうとすれば EMS を使わなければならず，0 から 4G−1 までのアドレス空間が直線的にあることを前提にしてつくられた UNIX の C プログラムに，かなり手を加える必要がでてくるからです。

□ DOS エクステンダによるメモリ拡張

MS-DOS でこうした大規模アプリケーションを動かしたい，という要求に

こたえてくれるのが**DOSエクステンダ**です(p.109を参照)。DOSエクステンダはMS-DOSからプロテクトモードプログラムを起動し，プログラム終了後，またMS-DOSに戻ってくるということを可能にしてくれます。

DOSエクステンダを使ったプロテクトモードプログラムは，MS-DOSの画面やファイルシステムなどを利用することができるので，たいへん便利です。プロテクトモードプログラムがMS-DOS用の資源を利用する際にはDOSエクステンダが仲立ちになります。

386のプロテクトモードプログラムは，最大セグメントサイズが4Gバイトであるために，32ビットアドレスで扱える範囲のどんな巨大なコード，データに対しても複数のセグメントに分割しなければならない，という事態が生じません。そのため，プロテクトモードプログラムの主要なコードセグメント，データセグメントはひとつずつで間に合います。

現在，日本で手に入る386用DOSエクステンダというと，Phar Lap社の386|DOS-Extender，京都マイクロコンピュータ社のexe386，また，IBM-PC用にD.J.Delorie氏が移植した「gcc」と「g++」であるdjgppに付属の「go32」などがあります(PC-9801版もある)。このうち，前二者のDOSエクステンダは，拡張子〝.EXP〟が付くプロテクトモードプログラム形式をサポートしています。この〝.EXP〟形式の実行ファイルは，MetaWare社の「High C」コンパイラが生成します。djgppに付属するDOSエクステンダは，やはりdjgppに付属するフリーソフトウェアのgccコンパイラが生成する〝a.out〟形式の実行ファイルをサポートします。

どちらのDOSエクステンダを使う場合でも，プロテクトモードプログラムの開発の流れは同じです。つまり，Cプログラムソースをプロテクトモード対応Cコンパイラでコンパイルし，DOSエクステンダ対応のライブラリとリンクします。そして，MS-DOSからできた実行ファイルを，DOSエクステンダとともに実行します。

4.3　386|TOOLBOX

先ほど挙げた開発ツールについて，もう少し説明をしましょう。先に紹介し

◉ 386|DOS-Extender システムコール(INT 21H)

機能番号	機能名
2501H	386|DOS-Extender データ構造のリセット
2502H	プロテクトモードの割り込みベクタの取得
2503H	リアルモードの割り込みベクタの取得
2504H	プロテクトモードの割り込みベクタのセット
2505H	リアルモードの割り込みベクタのセット
2507H	プロテクト/リアルモードの割り込みベクタのセット
2506H	常にプロテクトモード下で動作する割り込みのセット
2508H	セグメントのリニアベースアドレスの取得
2509H	リニアアドレスから物理アドレスへの変換
250AH	セグメントの最後への物理メモリのマッピング
250CH	ハードウェア割り込みベクタの取得
250DH	リアルモードとのリンク情報の取得
250EH	リアルモード手続きの呼び出し
250FH	プロテクトモードアドレスから MS-DOS のアドレスへの変換
2510H	レジスタ指定付きのリアルモード手続きの呼び出し
2511H	レジスタ指定付きのリアルモード割り込みの実行
2512H	デバッグ用のプログラムロード
2513H	セグメントディスクリプタのエイリアス
2514H	セグメント属性の変更
2515H	セグメント属性の取得
2516H	LDT が占めるメモリの解放
2517H	DOS のデータバッファに関する情報の取得
2518H	移動したセグメントに対するハンドラの指定
2519H	メモリエラーに関する拡張エラー情報
2520H	メモリの状態に関する情報の取得
2521H	拡張メモリの使用制限
2525H	1M バイト以下の従来のメモリの使用制限
25C0H	MS-DOS のメモリブロックの割り当て
25C1H	MS-DOS のメモリブロックの解放
25C2H	MS-DOS のメモリブロックのサイズ変更
25C3H	プログラムの実行

たPhar Lap社はDOSエクステンダの老舗で，ここから発売されている386|TOOLBOXは，もっともメジャーなDOSエクステンダであるといえるでしょう。フルパッケージの386|TOOLBOXを購入すれば，

386|ASM（アセンブラ）
386|LINK（リンカ）
386|LIB（ライブラリアン）
RUN386（386|DOS-Extender）
MINIBUG（デバッガ）
CFIG386（カスタマイズプログラム）

といった，プロテクトモードプログラムの開発に必要な一式と，ドキュメントが利用できます。もちろん，これで開発したプロテクトモードアプリケーションを，商品として販売するためのライセンスも用意されています。富士通のFM-Townsのシステムが，386をネイティブモードで動かすために386|DOS-Extenderを使っていることはよく知られていることです。また，開発されてから十分な時間が経っており，すでに市販のアプリケーションで，386|DOS-Extenderを使って動くものがあることからも，この製品が非常に信頼性の高いものであると伺えます。

また，386|DOS-Extenderは，386EMSエミュレータと共存するための，VCPI規格設立の発端ともいえるものです。最近では，プロテクトモード環境（OS）でのDOSエクステンダ規格ともいえるDPMIに準拠したものが発売されています。

4.4 High C

MetaWare社のHigh Cは，386パソコンでは非常に定評のあるコンパイラです。『C MAGAZINE』（1990年8月号）が行ったベンチマークテストによると，リアルモード用のMicrosoft C Ver.6.0に比べて，プロテクトモード用のMetaWare社のHigh C Ver.1.6は，

Dhrystoneで1割
Whetstoneで2割から5割

というように，高い性能を示しています。このことからも，非常に高い技術でコード最適化を行っていることがわかると思います。

High C Ver.1.6の最適化は，おおよそ次の通りです。

- **共通部分式の削除**
- **レジスタ内容の保持と再利用**
- **不用コードの削除**
- **ジャンプ命令サイズ最小化**
- **定数のたたみ込み**
- **定数とコピーの伝搬**
- **高速なプロシージャコール**
- **数値強度リダクション**
- **ループからの不変式の移動**
- **変数のレジスタへの自動マッピング**

また，生成するコードの安定性(コンパイラのバグの少なさ)も優れています。

4.5 フリーソフトウェアのCコンパイラ

□ gcc

Free Software Foundation Inc.のgccは，世界中のさまざまな研究者によって開発されている，フリーソフトウェアのCコンパイラです。したがって，入手する手間はあるものの，無料です。ただし，著作権などに対して細かい注意が必要です。

一般に，gccのようなフリーソフトウェアを使う際，肝に命じなければならないのは，バグがあってもすぐに直るとは限らない，ということです。何回もバージョンアップを繰り返すうちに，ユーザから届いたバグレポートが徐々に反映されていきますから，世界各地の多くの人々の努力によってバグがなくなっていくという感じです。製品として購入したコンパイラを使用する場合より

もリスクは大きいのですが，gccは移植性が高いので，多くのCPUに移植されており，ユーザ層が非常に広範にわたっています。それだけバグの洗い出しも早かったわけで，現在はかなりの場合で問題がないといえるでしょう。

gccは，Ver.1で細かなバージョンアップを繰り返したあと，最近大きなバージョンアップがあり，Ver.2.1になりました。

さて，gccには，最近のコンパイラ技術をふんだんに取り入れられています。また，複数のCPUに対応するための仮想化も行っています。gccでは，C言語のソースをコンパイルするときに，CPUに依存しないRTLと呼ばれる中間言語にいったんおとします。そして，さらにそのRTLから，今度はCPUに依存したコードにおとします。CPUから独立しているRTLを，CPUのアセンブリコードに変換するための記述は体系化されており，こうした技術が一体となって，gccの移植性を高めています。

RTLレベルでは，さまざまなコードの最適化を行います。そのためRTLという1コードでの最適化プログラムを書けば，gccが移植される多くのCPUに，その恩恵が与えられるという点で優れています。

しかし，こうしたCPUを仮想化した中間言語の生成は，複数のCPUをターゲットにしているコンパイラメーカーなら当然もっている技術であり，UNIXワークステーションに付属のCコンパイラと比較すれば，gccが決して優れた性能を示してくれるとは限りません。むしろ，製品化されたCコンパイラのほうが，たいていの場合，まだ優れています。とはいえ，gccにはターゲットマシンの記述をする部分があり，そのでき具合によっても性能は大きく向上します。

回 386用gcc

gccが動作する環境の多くは，UNIXです。

UNIXワークステーションのアドレッシングはリニアアドレスで，セグメンテーションを利用したものはあまりない，といってもいい状況です。386は，プロテクトモードではコードセグメント，データセグメント，スタックセグメントをプロセス空間と対応させることができます。こうすると，セグメントオフセットによるアクセスは，リニアアドレスによるアクセスと同じになり，非

セグメンテーションの CPU と同じコンパイラ技術を利用できます。そこで，gcc の発展と 386 の登場がうまくかみ合って，386 用の gcc がつくられました。MS-DOS 上で動作する実行ファイルの gcc は，djgpp(D.J.Delorie 氏の gcc と g++)パッケージに入っています。

そもそも 386 プロテクトモードのコードを gcc が生成することと，DOS エクステンダの存在は，直接には関係ありません。なぜなら，DOS エクステンダは，MS-DOS の環境でプロテクトモードコードが走るために橋渡しをするためのものだからです。

◻フリーソフトウェアの DOS エクステンダ「go32」

go32 は，djgpp パッケージに入っている DOS エクステンダです。gcc でも触れたように，djgpp は MS-DOS 上で動作するようにつくられた GNU C，GNU C++のコンパイル環境で，D.J.Delorie 氏が作成したものです。パッケージの内容を大まかに紹介すると，

- **32 ビット 386DOS エクステンダ(＋シンボリックデバッガ)**
- **C/C++コンパイラ(＋ユーティリティ)**
- **ライブラリ**
- **ソース**

となっており，32 ビットプロテクトモードプログラムを作成します。中でも，go32 は，**プロテクトモードプログラムの仮想記憶化**を実現することが特徴で，現在，VCPI まで対応しています。386 のページング機能を生かした DOS 用の仮想記憶制御ソフトはほとんどない状況なので，それなりのインパクトはありました。

ただし，DPMI に対応していないので，Windows 3.0 の 386 エンハンストモードの DOS ボックス上で，DOS アプリケーションとして動作させることはできません。したがって，Windows を中心に DOS 環境を構築している人は，go32 を使った JTeX や demacs などのツールを使う場合には，いったん Windows を抜けなくてはなりません。

djgpp のようなフリーソフトウェアとして広く世界中に配布されているものは，数多くのユーザの中からサポートしてくれる人が充実すれば，足りない点

は徐々に満たされていくものです。いま現在DPMIホストとなるものが，代表的なものとしてWindowsくらいなので，DPMI対応も後回しになった形ですが，いずれ徐々にサポートされるでしょう。

回シンボリックデバッガ

D.J.Delorie氏のMS-DOS用のDOSエクステンダのパッケージには，gccでコンパイルした実行ファイルをデバッグするためのツールも含まれています。シンボリックデバッガのdebug32です。

debug32は，DOSエクステンダであるgo32を多少書き換えたもので，DOSエクステンダアプリケーションと同様に386/486上のMS-DOSから起動し，デバッグを行います。デバッグのコマンドは，シンボリックデバッガに標準的なものがだいたい揃っており，使われるに従って追加されていく可能性もあります。

debug32のブレークポイントの設定の仕方は，386のハードウェアデバッグ機能と関連しています。デバッグレジスタについては，0から3まで4つあることをデバッグサポート(p.96)でも説明していますが，このデバッグレジスタにアドレスを設定すると，

- **そのアドレスの命令の実行で停止する**
- **そのアドレスへのアクセスで実行を停止する**

ことができます。debug32では，このうちの前者だけを利用し，後者は利用されていないようです(確認した時点では)。後者も利用できれば，特定のメモリアドレス(変数)へのアクセスだけを監視することができるでしょう。ブレークポイントを使用する各サブコマンドは，次のようなものです。

go	ブレークポイント3	
step	INTO	
	next	ブレークポイント3
	break	ブレークポイント0～3

go，next，breakはブレークポイント3を共用する

5 386|DOS-Extender Ver.4.1

この節では，Windows 3.0の**386エンハンストモード**で，DOSエクステンダアプリケーションを使うことを紹介します。このモードでは，386の機能を生かして複数のDOSを起動することができます。そうしたDOSは，Windows環境ではない，従来のMS-DOS環境と同様に使えるので，DOSエクステンダアプリケーションがWindowsとともに使用できます。ここで，ぜひWindows 3.0のもとでの32ビットプロテクトモードプログラミングの雰囲気を感じてください。

5.1 386エンハンストモードの環境

Windows 3.0は，DOSの“Look & Feel”(見た目と使い心地)を大幅に改善しています。Windowsのそうした点も重要ですが，ここでは386エンハンストモードで実現されたマルチDOSに重点をおいて解説をすすめます。

386エンハンストモードのWindowsでは，CPUの仮想86モードを使ってマルチDOSを実現しています。つまり，Windowsで起動されるDOSが386アーキテクチャで，次のように実現されていることを意味します。

(1)プロテクトモードタスクとして独立したアドレス空間(独自のPTE)
(2)仮想86モードで動作
(3)マルチタスクとしてWindowsがTSSのスケジューリングを管理

(1)は,「独立したアドレス空間」を各DOSにもたせることによって，仮に複数起動されたDOSアプリケーションのひとつが暴走しても，ほかのDOSやWindowsへの影響を最小限に抑えられます。これは複数のDOSを起動できる

ための重要な要素です。

(2)の**仮想86モード**は，仮想86マシンを実現して，DOS上で動いているアプリケーションにWindowsであると意識させずに，DOSマシン上で動いているかのように見せるものです。

また，Windowsがリング0で動作するのに対して，DOSは仮想86モードのためにリング3で動作しますが，それをDOSエクステンダで純粋プロテクトモードにすれば，リング1で動作します。

(3)のWindowsのDOSタスクスケジューリングは，プリエンプト方式です。つまり，DOSタスクは与えられたタイムスライスだけの時間を実行したら，その実行は打ち切られ(プリエンプト)，Windowsは別のタスクの実行をします(プリエンプトについての詳細は，第2章の「2　マルチタスク」p.41を参照)。DOSアプリケーション切り替えのプリエンプト方式に対して，Windowsアプリケーション間の切り替えは，イベントで切り替わる方式です。ある瞬間動作しているWindowsアプリケーションから，ほかのWindowsアプリケーションへの切り替えは，動作しているWindowsアプリケーションが実行を何らかの形で放棄しないと生じません。したがって，「行儀の悪い」Windowsアプリケーションが実行権を握って離さないと，ほかのWindowsアプリケーションは動かなくなってしまいます。その点，DOSに対してWindows 3.0では，Windowsアプリケーションは全部でひとつのタスクと考えられ，プリエンプトされてからほかのDOSタスクに動作が移るので，行儀の悪いWindowsアプリケーションがいてもDOSには影響ありません。

しかし，仮想86モードによってマルチタスク化しただけでは，マルチDOSになるわけではありません。それぞれのDOSが表示画面をもちますが，それを同じ画面に表示するのでは，マルチタスクもメリットはありません。キーボードも同様で，キーボードからの入力を特定のアプリケーションに送れるようにできなくてはいけません。Windowsのアプリケーションは，仮想デバイスドライバによる共有リソースの仮想化(キーボード，画面など)の標準化を考えてあるので問題はないのですが，もともとDOSアプリケーションは1台のマシンを占有して動くものですから，そのような仮想化をアプリケーションでは，まったく考慮していないのです。そこでWindowsは，マルチDOSとして描画の標準システムコールに対しては，標準システムコールのすげ替え(**フック**)

を行います。画面のある場所に文字を表示するシステムコールなら，それをDOS のウィンドウの仮想画面表示に置き換えます。プログラムがアドレスを指定して VRAM に直接描くようなアプリケーションでも，仮想 86 モードで VRAM の領域に仮想画面用の RAM をマッピングできるので，画面の仮想化が可能です。しかし，CPU 以外(たとえば GDC のようなグラフィックプロセッサ)が，VRAM に直接アクセスするような場合は仮想化ができないので，そういったアプリケーションの場合は画面を全部ゆずって(フルスクリーンにして)動かします。逆に，ウィンドウ表示にする場合はフルスクリーンを待避して，ウィンドウ表示を復帰させます。

◻ DOS ボックスから 386|DOS-Extender を起動

Windows から 386|DOS-Extender アプリケーションを起動する方法は，DOS アプリケーションを起動する方法と同様に，次のような方法があります。いずれの方法でも，386|DOS-Extender アプリケーションは仮想 86 モードで起動され，その後プロテクトモードに切り替わります。

(1) DOS プロンプトウィンドウの〝COMMAND.COM〟から起動
(2)アプリケーション専用のプログラムマネージャアイテムによる起動
(3)〝.EXP〟ファイルと〝RUN386.EXE〟との関連付けによる起動

(1)は，通常の DOS 環境である〝COMMAND.COM〟のプロンプトから起動する方法と同じ使い勝手です。

(2)は，Windows のアイコンによる起動方法で，(1)の方法をマウス操作だけでできるようにした方法です。

(3)は，Windows で DOS に追加された，データ指向的なアプリケーション起動方法です。データ(ファイル)を中心に考え，データを操作するプログラムはデータファイルの属性のひとつとして見ます。この場合では，〝.EXP〟という拡張子が付いたファイルは必ず〝RUN386.EXE〟が処理する，と Windows 内で決めます。すると〝.EXP〟ファイルを起動したとき，(1)および(2)の方法のように，〝RUN386.EXE〟がそのファイルを引き数として起動されます。(3)の方法では，386|DOS-Extender アプリケーションごとにプログラムマネージャアイテムをつくらなくてすむので，たいへん便利です。〝RUN386.EXE〟用の

PIF(Program Information File)をつくっておくと,〝RUN386.EXE〟がWindowsで起動される場合に,メモリをどのくらい使うか,マルチタスクの優先度,画面をどのように使うか,などの条件を設定できます。

こうして起動された386|DOS-Extenderアプリケーションは,Windows中の1DOSアプリケーションとみなされ,仮想86モードの1タスクとして実行されます。PIFによって適切な条件選択が行われていれば,生成されたタスクはプリエンプトされるマルチタスクの環境で実行されます。また,たとえバックグラウンドになったとしても,Windowsのスケジューラによってタイムスライスが与えられ,実行され続けます。

Windows, つまりDPMI Ver.0.9ホストからもらうLDTは,実はひとつしかありません。そのため,すべてのプロテクトモードアプリケーションは,同一のLDTを使って動きます(LDTも共有します)。DPMI Ver.0.9では,同一リニアアドレス空間上ですべてのDOSプロテクトモードプログラムが動くことになり,例外ハンドラも同じものを使用します。DPMI Ver.1.0では,クライアントごとに別々のLDT, IDTをもつことができます。Ver.0.9でも,異なるプロテクトモードは別々のTSS, つまりページ変換テーブルをもち,リニアアドレスの使い方が同じでも別々の物理メモリを使うことを思い出してください。

5.2 仮想デバイスドライバの利用

ところで,Windows 3.0は,DPMI Ver.0.9のINT 31Hコールをサポートしていますが,拡張性のためにそれ以外のINTコールも後付けでサポートする機能があります。それが仮想デバイスドライバと呼ばれる,32ビットプロテクトモードの**ダイナミックリンクライブラリ(DLL)機能**です。

Windowsの〝SYSTEM.INI〟ファイルの386エンハンストモード用の「386enh」部に,デバイスドライバを指定すると,Windows起動時にデバイスドライバとして登録され,生のWindowsがサポートしないINTコールを含む割り込み/例外の処理ハンドラとして機能します。この機能は,386|

DOS-Extender Ver.4.x アプリケーションを Windows 3.0 で動作させようとした場合に，とても重要なものです。

386|DOS-Extender Ver.4 は DPMI に対応していますが，Windows 3.0 の DPMI ホストはバージョンが 0.9 であることもあって，386|DOS-Extender に必要なリソース管理能力をすべてもっているわけではありません。かといって，低い特権レベルで動作しているアプリケーションは，Windows が管理すべきリソースを操作できません。そこで必要となるのが，386|DOS-Extender Ver.4 やその開発環境が Windows 3.0 下で使用するデバイスドライバ〝PHARLAP.386〟です。

これは，先ほど述べた Windows がシステムの一部として組み込む，仮想デバイスドライバのひとつです。〝PHARLAP.386〟は，Windows 3.0(DPMI Ver.0.9) がサポートしていない，一部のプロテクトモードマルチ DOS アプリケーションが必要とする，

(1)デバッグレジスタの操作と仮想化
(2)コンベンショナルメモリのマッピング用の低レベル機能
(3)数値演算コプロセッサ機能

を提供します。(1)は，共有リソースであるデバッグレジスタを，Windows 下でデバッガが使うためのものです。(2)は，Windows 下で，リアルモードとプロテクトモードの混在コードである 386|DOS-Extender アプリケーションを実行させるためのものです。

5.3 386|DOS-Extender で仮想記憶の実現

Windows 3.0 の 386 エンハンストモードでは，**仮想記憶**システムをもっています。そのため，386 エンハンストモードの DOS ボックスから起動する DOS エクステンダプログラムは，自動的に仮想記憶で動作します。

しかし，1 アプリケーションが使えるデフォルトの仮想記憶は制限され，DOS ボックスにおけるプロテクトモードアプリケーションもその対象になり

ます。デフォルトの仮想記憶を制限するのは，むやみに大きな仮想記憶を使うプログラムを許すと，そのプログラムの実行が物理メモリとディスクとの間のデータ転送を増やして，Windows 全体の性能を落とすからです。

386 エンハンストモードにおいて，Windows アプリケーションは，2.1M バイト(『日本語 Microsoft Systems Journal』1992 年 4 月号の実験記事)の，また 386|DOS-Extender アプリケーションは 2M バイト(Phar Lap 社の資料)の動的メモリブロック確保に成功しています。どちらも積んでいる物理メモリの量に関係なく(より多く)確保できています。しかし，せっかく Windows の仮想記憶システムで物理メモリ以上にメモリを使えるというのに，DOS プロテクトモードプログラムへの制限が 2M バイトというのは不親切です。

8086 パソコンの DOS 環境で 386|DOS-Extender は，プロテクトモードプログラムが malloc() 関数などで動的にアクセスするメモリ領域を増やすと，空き物理メモリがある限り，必要に応じて 128K バイト単位で物理メモリをマップしていきます。つまり，データセグメント中，物理デバイスにマップされたページが動的に増えるようになっています。これと対照的に 386 エンハンストモードの DOS ボックスでは，386|DOS-Extender はページを動的にマップして増やすことができません。プログラム実行中にデフォルトとして与えられる 2M バイトのデータセグメントを拡張できないのです。

この理由は，次の通りです。

マップ済みメモリブロックのサイズ変更は，DPMI の INT 31H AX=0503H によって行われますが(malloc() 関数が，386|DOS-Extender を通じて呼び出す)，その際，そのメモリブロックの先頭リニアアドレスが変わる可能性があります。たとえばサイズを増やしたいのにリニアアドレス上，その前後にほかのメモリブロックがあるとき，まったく別の場所にメモリブロックを移してサイズを増やすことがあります(ページング機能を使えば，マッピングを変更するだけです)。仮想記憶を使わない DOS 環境では，ディスクリプタのセグメントベースを変更すれば，同じ「セレクタ：オフセット」で同じ場所(物理アドレス)にアクセスできます。しかし，Windows 3.0 の仮想記憶システムでは同じことをやっても，変更前にサイズ変更を受けるメモリブロックのページがディスクに退避されていると，先頭リニアアドレスが変わってしまったメモリブロックのページを Windows が物理メモリに正しく読み込みません(偶然，先頭リニア

アドレスが変わらなければ，問題ありませんが)。

つまり，Windows 3.0 では，データセグメントのサイズ変更のために DPMI ファンクションを使えないことになります。386|DOS-Extender アプリケーションが，386 エンハンストモードのデフォルトより大きなデータセグメントを確実に得るためには，起動時に必要な大きさのセグメントを指定して確保してしまいます。そうすれば 2M バイトに留まらず，許される限りのサイズのデータセグメントをもつ，プロテクトモードプログラムをつくることが可能になります。

5.4 Windows 3.0 と DOS エクステンダプログラミング

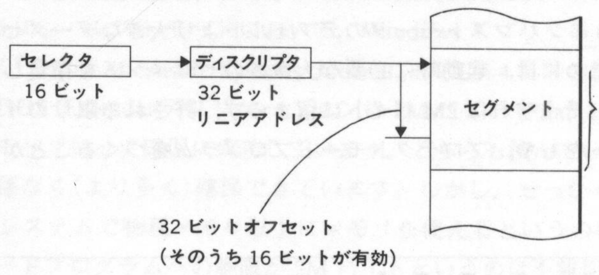
□ Windows の 16 ビット API

8086，80286 でのアドレスは，セグメント指定を除けば 16 ビットオフセットで指定されるため，最大でも 64K バイトのセグメントしか使えませんが，386 では 32 ビットオフセット指定が可能であり，最大 4G バイトのセグメントを使えます。そのため，386 以前の 86 系アーキテクチャでは連続したアドレスでアクセスできるデータは 64K バイトまでであり，セグメントをまたぐポインタ操作はやっかいで，しかも性能を落とすものでした。

Windows 3.0 では，386 以上にターゲットを絞った 386 エンハンストモードでも，標準の 32 ビット API(Application Program Interface)はなく，16 ビット API です(fig.3.6)。そのために 386 で Windows 3.0 を使う場合でも，標準 API で扱うひとつのセグメントの最大のサイズは 64K バイトです。したがって，1M バイトを超えるようなセグメントを扱う場合は，互換性を重視すれば，64K バイトセグメントの組み合わせで実現するか，Windows SDK に付属する 386 エンハンストモード用ダイナミックリンクライブラリ〝WINMEM32.DLL〟を利用します。ただし，将来の Windows で提供されるであろう標準 32 ビット API が，〝WINMEM32.DLL〟と同じになるかどうかはわかりません。

誤解のないようにしておきたいのですが，386 エンハンストモードでの

● fig.3.6 ● Windowsの16ビットAPIによるセグメント



Windowsアプリケーションは仮装記憶システムを使って，搭載している物理メモリ以上のメモリを割り当てることができます。リニアアドレスは32ビットで生成されます。しかし，現在与えられているAPIでは，割り当てられるひとつひとつのセグメントのサイズは16ビットで，アクセスできるもは64Kバイトまでなのです。初めから386をターゲットにしたWindowsの386エンハンストモードであっても16ビットAPIなのは，80286の標準モードとの互換性をとるためです。

ただし，近い将来には，Windowsの32ビットAPIが登場することでしょう。32ビットのウィンドウアプリケーションがつくられれば，386本来の性能を生かし，データセグメントなどが64Kバイトに制限されるようなことはありません。32ビットAPIによるウィンドウアプリケーションについての言及は，実物が登場してからということにしたいと思います。

さらにいえば，386エンハンストモードでは，Windowsアプリケーションが LDTを共有しています。LDTは最大でも64Kバイトの大きさしかなく，1ディスクリプタあたり8バイト占めることから，Windowsアプリケーションは合計で8,192個のセレクタ(13ビット分)しかもてません。仮想86モードを使ったWindowsの仮想DOSマシンや本来のDOSマシンは，セグメントディスクリプタを使わないためにそうした制限はなく，16ビット分，64K個の種

類のセグメント指定をできます。Windowsでセグメント単位にメモリを割り当てていると，ディスクリプタがつきてしまうことは，すでに指摘されています。そうしたディスクリプタの枯渇を避けるためには，64Kバイトのセグメント内で，さらに細かいメモリ管理を行う必要があります。

◻ DOSエクステンダの32ビットリニアアドレッシング

Windows環境でプログラミングを必要とする人の誰もが，Windowsアプリケーションを書くわけではありません。自分専用に書くプログラムでは標準入出力とファイルしか使わないという場合が，まだまだ圧倒的に多いのではないでしょうか？　Windowsのユーザインターフェイスをもつアプリケーションをつくることがどんなに楽になっても，仮想DOSマシンがサポートされる限り，自分用には仮想DOSマシン(つまりDOS)上で動くプログラムを書く場合が，大部分だと思います。

また，現在DOS用のアプリケーションを使ったほうが有利な面もあります。それはDOSエクステンダによって，**32ビットリニアアドレッシング**のプログラムが組めることです。このメリットは，仮想記憶とともに1Mバイトを超えるプログラムを，プログラム自体の性能をそれほど落とさずに，楽につくれることです。しかも，プログラムが複雑でなくても，1Mバイト以上のデータを使いたいプログラムは，たとえ個人用であっても存在します。16ビットと比較すると，32ビットプロテクトモードのほうが，単に大きなデータを使えるばかりでなく，性能上も好ましい点があります。16ビットでは，セグメントの境界を越える場合のアクセスについては，HUGEポインタのようにセグメントの切り替わりのある，あるいはありそうなポインタ操作のたびに，どのセグメントへ切り替わるのかチェックして切り替えるコードが入ります。これは32ビットリニアアドレッシングから見れば余分なコードといわざるをえず，性能上も感心できません。

5.5　クリップボードを使った通信

どんなOSの下でも動作するようなプログラムがあれば，反対に，動作して

いる環境に依存したプログラムもあります。これは見方を変えていえば，動作している環境の特性を生かした，凝ったプログラムをつくる場合があるといえます。その典型的なものがシステムのファンクションコールです。

ファンクションコールは，DOSであればINT 21Hなどで呼び出されるシステムサービスルーチンのことで，いってみればシステム固有のサブルーチンのことです。

DOSの誕生からいままでに，さまざまなDOS関係のファンクションが提供されてきました。EMSが開発されるとEMSファンクション，Windowsが開発されるとWindowsファンクション，といった具合に……。

リアルモード，いわゆる通常のDOS環境下のDOSアプリケーションが，こうしたDOSまわりに提供されているWindows固有のサービスやEMS固有のサービスなどを利用すれば，ほかのシステムへの移植性は落ちますが，環境を最大限に生かしたプログラムづくりが可能になります。

ところで，こうしたDOSまわりの環境は，リアルモード中心に開発されてきました。つまり，ファンクションコールで使用するINT割り込みや必要になるバッファやポインタは，すべてリアルモードのものが指定/使用されており，プロテクトモードには提供されていません。

それでは，プロテクトモードアプリケーションは使えずに孤立するのでしょうか？　いいえ，そのようなことはありません。このようなDOSの基本機能を拡張するリアルモードファンクションは，DOSエクステンダによるプロテクトモードアプリケーションが使用することで可能になっています。これについては，このあとにWindowsのクリップボードファンクションを例にして具体的な説明をしましょう。クリップボードでは，386エンハンストモード上で動作する

- 386|DOS-Extender Ver.4アプリケーション
- Windowsアプリケーション

同士の通信を考えます。なぜなら，クリップボードは，p.124でも述べたように，Windows上で走るすべてプログラムの共有セグメントのような存在だか

らです。

ここで重要なのは，DOS エクステンダを通じてプロテクトモードアプリケーションが，リアルモードのファンクションを簡単に利用することができるということです。このことで，プロテクトモード DOS アプリケーションにも，リアルモードの DOS アプリケーションと同じ使い勝手を望むことができるわけです。

回クリップボード

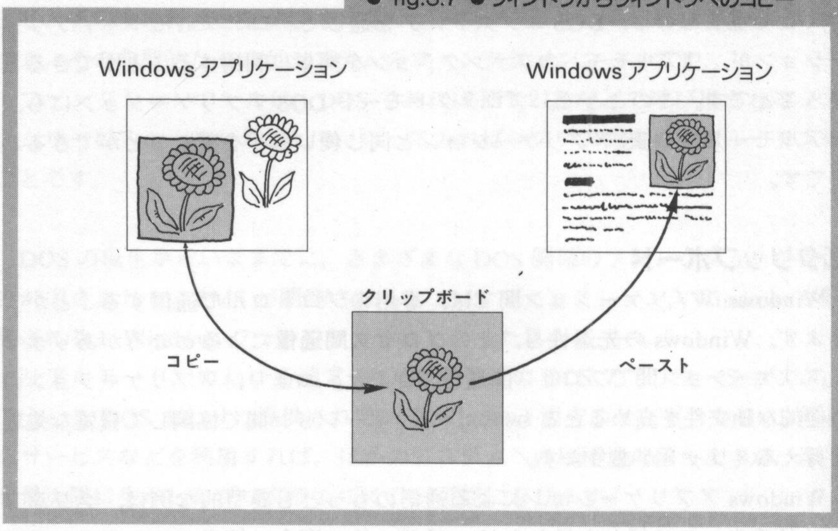
Windows アプリケーション間では，専用のプロトコルで通信することができます。Windows の先進性も，その**プロセス間通信**によるところがあります。アプリケーション間でプロセス間通信できると，つまり，アプリケーションの機能的な独立性を高めるとともに，アプリケーション間で協調して複雑な処理を行えるメリットがあります。

Windows アプリケーションによる通信のもっとも典型的な例は，クリップボードです。ユーザから見たクリップボードの概念は簡単です。では，ある Windows アプリケーションウィンドウから別の Windows アプリケーションへ，絵や文章などのオブジェクトをコピーする場合を考えてみましょう。

Windows では，コピー元のウィンドウでコピーしたいオブジェクトを選択し，COPY コマンドをクリックします。それをコピー先のウィンドウで「ペースト」すると，先に別ウィンドウで選択コピーしたオブジェクトがペーストされ，結果としてコピーが完了します(fig.3.7)。

このコピーの仕組みでは，クリップボードという Windows システム全体で知られている「バッファ領域」が重要な役割を担います。コピー元のウィンドウでコピーされた絵などのオブジェクトは，いったんクリップボードにコピーされます。この時点では，コピー元のウィンドウ(アプリケーション)は誰にコピーするのかわからないので，とりあえずクリップボードにコピーするのです。クリップボードにコピーしたあと，このアプリケーションはコピーに対して何の責任ももちません。そして，続けてコピー先のウィンドウでペーストを行うと，クリップボードの内容を再度ペーストします。これも，誰からオブジェク

● fig.3.7 ●ウィンドウからウィンドウへのコピー



トがコピーされるか，ということに関係なく行うことができます。アプリケーション間のオブジェクトのコピー＆ペーストはこうした，

- **●アプリケーションからクリップボードにコピー**
- **●クリップボードからアプリケーションにコピー**

というクリップボードを介したふたつの手続きで構成されているのです。

回リアルモードファンクションコール

オブジェクトをクリップボードに置いたり，クリップボードからオブジェクトをとってきたりするクリップボードへのアクセスは，DOSコールのINT 2FH AH=17Hで行われます。DOSファンクションコールはリアルモードです。そのため，DOSエクステンダアプリケーションがクリップボードにアクセスするには，リアルモードのファンクションコールを発行できる必要があります。

32ビットプロテクトモードプログラム用のDOSエクステンダは，DOSのリアルモードファンクションコールの機能を提供する，プロテクトモードファ

ンクションコールをサポートしています。この仕組みは，p.111 で説明しているように，プロテクトモードで発行されたファンクションコールを DOS エクステンダが受けて，DOS ファンクションコールを発行し直す方法で実現されています。しかし，クリップボードのためのファンクションコール INT 2FH API は，標準の DOS コールでないこともあって，この対象にはなっていません。

こうした DOS エクステンダがサポートしていないリアルモードファンクションコールを，プロテクトモードから発行するために 386|DOS-Extender では，「リアルモード割り込みを発行する」というプロテクトモードファンクションコールがあります。詳細は次の通りです。

INT 21H
AX = 2511H

入力

DS：EDX　　パラメータブロックへのポインタ

出力

セグメントレジスタ　　不変
EDX　　不変
その他　　リアルモード割り込みハンドラによる

パラメータブロックの構造

オフセット	サイズ	説　明
00H	WORD	発行する割り込み番号(不変)
02H	WORD	リアルモードの DS
04H	WORD	リアルモードの ES
06H	WORD	リアルモードの FS
08H	WORD	リアルモードの GS
0AH	DWORD	リアルモードの EAX(不変)
0EH	DWORD	リアルモードの EDX

プロテクトモードプログラムが，INT 21H ファンクション 2511H を発行すると，DOS エクステンダが制御を受け取り，リアルモードに切り替わってパラ

メータブロックで指定されたレジスタ値をレジスタにロードし，それから指定された番号のINTを指定されたファンクション番号で行います。リアルモードファンクションコールが終わると，セグメントレジスタとEDXの内容はパラメータブロックに退避されたのち，元のセグメントレジスタ(セレクタ値)とEDXが復帰します。そのほかのレジスタ値は，リアルモードでセットされた値のままです。

回クリップボードAPI

386エンハンストモードのクリップボードAPI(INT 2FH AH=17H)には，次のようなファンクションがあります。

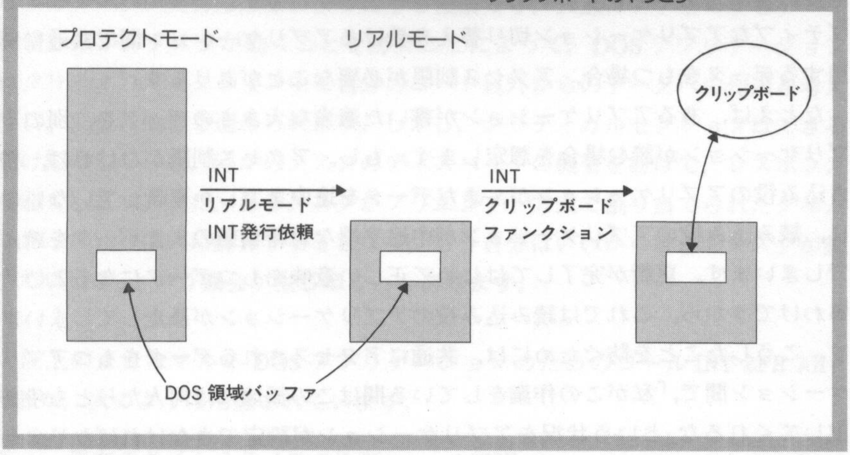
IdentifyWinOldApVersion	1700H	
OpenClipboard	1701H	
EmptyClipboard	1702H	
SetClipboardData	1703H	クリップボードにデータをセット
GetClipboardDataSize	1704H	
GetClipboardData	1705H	クリップボードからデータを得る
CloseClipboard	1708H	
ClipboardCompact	1709H	
GetDeviceCaps	170AH	

回リアルモードバッファ

DOSエクステンダのプロテクトモードプログラムは，データを32ビットオフセットで管理しています。しかし，クリップボードAPIはリアルモードのファンクションなので，データはリアルモードでアクセスできる1Mバイトに格納されていなければいけません。こうしたことのために，DOSエクステンダはリアルモード領域のバッファをつくる機能をもちます。

「MS-DOSメモリブロックを割り当てる」ファンクション25C0Hを使うと，最初の1Mバイトに領域を確保します。このDOSエクステンダファンクションは，成功すると16ビットのリアルモードパラグラフアドレスを返します。DOSエクステンダでは，セレクタ0037H(最後の2ビットはDPL)がDOS用

● fig.3.8 ● プロテクトモードプログラムとクリップボードのやりとり



の 1M バイトを 1 セグメントとして指しています。仮にファンクション 25C0H で得た DOS 領域のメモリブロック先頭の 16 ビットパラグラフアドレスを p とすると，プロテクトモードではセレクタ 0037H で指定されるセグメントのオフセット p<<4 (4 ビット左シフト) でアクセスできるわけです。

プロテクトモードプログラムは，この FAR ポインタ (セレクタ＝0037H，オフセット p<<4) を使って，クリップボードに置きたいデータを DOS 領域のメモリブロックに転送します。そして，リアルモードポインタ (パラグラフ p，オフセット 0) を使って，SetClipboardData ファンクションを，前に説明したリアルモードファンクションコールを使って発行します。このようにして DOS エクステンダアプリケーションからクリップボードにデータを置くことができます (fig.3.8)。

5.6 マルチタスク機能

386 エンハンストモードでは，DOS プログラムを複数起動することが可能です。起動したプログラムは，プリエンプティブに切り替わりながら動作しま

す。つまり，これは，動作中のアプリケーション自身が，それと気づかない間にほかのアプリケーションが動いている状況をいいます。そのため，プリエンプティブなアプリケーション切り替えを受けるアプリケーション間で共通に使用するデータをもつ場合，アクセス制限が必要なことがあります。

たとえば，あるアプリケーションが書いた適当な大きさのデータを，別のアプリケーションが読む場合を想定します。もし，アクセス制限がなければ，書き込み役のアプリケーションが，まだデータを途中までしか更新していないのに，読み込み役のアプリケーションが中途半端な更新状態のままデータを読んでしまいます。更新が完了してはじめて正しい意味をもつデータになるといえるわけですから，これでは読み込み役のアプリケーションが暴走してしまいます。こうしたことを防ぐためには，共通にアクセスされるデータをもつアプリケーション間で，「私がこの作業をしている間はこの領域には何人たりとも手を出してくれるな」という状況をアプリケーションが設定できなければなりません。そうした複数アプリケーション間の協調に差し障りのある作業を，**クリティカルセクションにあるコード**などといいます。そのひとつの方法には，プリエンプト方式で勝手に切り替わってしまうほかのアプリケーションへの切り替えを，アプリケーションが許可しないようにシステムに頼む方法があります。ほかのアプリケーションへの切り替えがなければ，データを書き込む役のアプリケーションがデータの更新を始める前に，ほかのアプリケーションへの切り替えが禁止され，更新途中のデータを読まれることを防ぐことができます。逆に，読み込み役のアプリケーションは，読む前にほかのアプリケーションへの切り替えを禁止すれば，読んでいる最中にデータが変更されることを防ぐことができます。ただし，この方法はアプリケーション間の協調をとる方法のほんのひとつにすぎず，よりきめの細かい方法はほかにもあります。しかし，その紹介は，本筋から外れてしまいますので，またの機会に譲ります。

さて，Windowsに話を戻すと，Windowsでも起動される複数のDOSアプリケーションが，共通なデータにアクセスする場合が考えられるでしょう。協調して正しく動作するためには，同期機構が必要となります。Windowsが提供するクリティカルセクションのための機構には，先ほど述べたアプリケーション切り替えを禁止する方法があります。これはWindowsに依頼されたクリ

ティカルセクションの開始と，終了の宣言の間のコードを実行している間は，ほかのタスクが実行できないようにする機構です。共通にアクセスするデータを使うほかのタスクが動くことを防ぐことによって，DOSアプリケーションはクリティカルセクション中に自分のコード以外からのデータへの作用を考えずに，安全に処理を進められます。しかし，クリティカルセクションはできるだけ短くしないと，ほかのタスクのディスパッチの機会を妨げて，レスポンスを悪くします。また，あるタスクがプリエンプト方式で割り当てられたタイムスライスいっぱいに計算することなく，「もう自分はいいからほかのタスクが動いていいよ」という機会が生じることもあります。

以上のようなマルチDOSアプリケーションのためのコールINT 2FH AH=16Hが，次のように用意されています。

現在のタイムスライスの放棄	1680H
クリティカルセクション開始	1681H
クリティカルセクション終了	1682H
現在の仮想マシンIDの取得	1683H

これらのファンクションコールは，クリップボードAPIであるINT 2FH AH=17Hをコールしたときとと同様にコールできます。つまり，プロテクトモードのDOSアプリケーションからDOSエクステンダを通じてリアルモードINT 2FH AH=16Hを発行することで，クリティカルセクションの設定やタイムスライスの放棄を行えます。

5.7 DPMIコール

DPMIコールは，386|DOS-Extenderを利用するプロテクトモードアプリケーションが必要とする機能であって，DPMIホストを直接呼び出さなくてはならないような機能はほとんどありません。DPMIホストが提供するプロテクトモードのINT 31Hインターフェイスによるファンクションは，DOSエクステンダよりも低レベルなもの(DOSエクステンダは，DPMIホストが存在すれば

その機能を利用する)になっています。DPMIファンクションをint86()関数などで直接コールするプロテクトモードアプリケーションを書くと，そのアプリケーションは，DPMIホストが存在していない環境では動作しないことになります(DPMIホストが提供する機能を利用することはできますが……)。DPMIをサポートする環境以外への移植性を重視するなら，当然DPMIホスト機能の直接コールはすべきではありません。

386エンハンストモードでしか使わない，あるいはDPMI Ver.1.0対応の環境でしか使わないという場合は，DPMI(p.116を参照)でリストアップされている機能を，プロテクトモードのアプリケーション(たとえば386|DOS-Extender)からのINT 31H(int86()，int86x())で呼び出せます。

5.8 DPMI Ver.0.9と386|DOS-Extender Ver.4

DPMI Ver.1.0に対応している386|DOS-Extenderが必要とする能力で，DPMI Ver.0.9でサポートされていないものをリストアップしておきます。つまり，Windows 3.0にはない機能になります。

◻ページング操作

DPMI Ver.0.9には，アプリケーションセグメント中にマップされていない論理ページを作成する機能がありません。386|DOS-Extenderは，アドレスオフセット0を指すポインタ(ヌルポインタ)を不正ポインタとして扱うことができます("−OFFSET"オプションスイッチ)。これはアプリケーションセグメントの最初の論理ページをページ不在にしておき，そのページへのアクセスをページフォールトにすることによって実現しています。DPMI Ver.0.9では，"−OFFSET"スイッチをサポートする機能をもちません。かといって，386|DOS-Extenderが直接操作することも，特権レベルによって保護されているためにできません。しかし，DPMI Ver.1.0ではファンクション0507Hでページ属性を変更することができます。

□物理デバイスマッピング

DPMI Ver.0.9には，ビデオメモリやコプロセッサといった物理デバイスを，アプリケーションが自分のセグメント中にマップする機能がありません。そのためDPMI Ver.0.9では，後述のような方法を提供して，386|DOS-Extenderが別のセグメントにマップした物理デバイスに，FARポインタでアクセスします。

DPMI Ver.1.0ではファンクション0508Hで物理デバイスに対応する物理アドレスを，以前に確保された指定したセグメントオフセットにマップできます。また，このコールでマップした物理デバイスには，オフセットの指定(NEARポインタ)だけでアクセスできます。ただし，このファンクションはオプション扱いです。

DPMI Ver.0.9にはファンクション0800Hという，物理デバイスのある物理メモリ領域をリニアアドレスの適当な領域(1Mバイトより上)にマップする機能があります。

アプリケーションがそこへアクセスするためには，そのリニアアドレスに対応する「セレクタ：オフセット」をつくる必要があります。したがって，このファンクションコールによってマップされた物理デバイスは，FARポインタでアクセスすることになります。しかし，この方式はアクセスにセグメントディスクリプタの切り替えを必要とし，性能上もファンクション0508Hのほうが優位です。どちらのファンクションにしても，DPMIホストは指定される物理アドレスからそれが仮想化すべきデバイスかどうかを判断し，仮想化されるデバイス(たとえばVRAM)のマッピング要求であれば，コールを拒否できます。

□コンベンショナルメモリ

386|DOS-Extenderでは，リアルモードコードとプロテクトモードコードが混在したアプリケーションをつくれます。こうしたアプリケーションは，ひとつのプロテクトモードセグメントの最初の64Kバイト以下に，リアルモードのコードとデータが入っています。アプリケーションリンク時に"-REALBREAK"スイッチを指定して，プロテクトモードセグメントの最初のどれだけをコンベンショナルメモリに連続に置くかをコードに埋め込みます。

通常のDOS環境で実行されると，リアルモードコードの部分はコンベンショナルメモリ内の物理的に連続な領域に，ひとつのセグメントとして配置されます。すると，リアルモード時には1Mバイト内に見えるセグメントとして見え，プロテクトモード時にはセグメントの最初の64Kバイト以下に見えます。そうなると，プロテクトモードからリアルモードコードのデータにアクセスできます。

こうした〝-REALBREAK〟スイッチによって，結果的にコンベンショナルメモリに置かれたセグメント(メモリ)をプロテクトモードセグメントの一部として参照できるようにすることは，DPMI Ver.0.9の機能ではサポートできません。そのため，仮想86モードでDOSに管理されている論理メモリの1Mバイト境界以下にマップされている物理メモリを，32ビットプロテクトモードリニアアドレス空間上に確保した領域にマップする必要があります。つまり，PTEエイリアスをつくる必要があります(エイリアスとは，ひとつの実体を指すためのものを複数つくること)。

DPMI Ver.1.0では，ファンクション0509Hでコンベンショナルメモリの指定されたリニアアドレス(20ビットが有効)から複数ページを，プロテクトモードアプリケーションセグメントの指定領域にマップできます。マップされるコンベンショナルメモリ(プロテクトモードコードに所有権が移っているページ)は，ファンクション0602Hコールでできるだけページ可能(ディスクへ移動可能)にしておきます。ファンクション0602Hは，リアルモード領域(リニア空間最初の1Mバイト)をページ可能にマークします。ファンクション0509Hは，DPMI Ver.1.0でもオプション扱いです。

Windowsのように仮想記憶システムをもっているDPMIホストでは，ファンクション0509Hのような**PTEエイリアス機能**はちょっと厄介です。それは次のような理由によります。つまり，エイリアスされている物理ページについては，そのひとつのPTEの操作がほかのPTEにも反映されなければならない場合があるからです。これを典型的なPTEへの操作である，ページ不在で考えてみましょう。

ひとつのPTEをページ不在にして該当ページをディスクにはきだせば，そのページへアドレス変換するほかのPTEもページ不在にしなければ，つじつまが合いません。しかし，つじつまを合わせるには，ディスクへページアウトする際に，

- **エイリアスされているかどうか調べ，**
- **エイリアスされていれば，関係するすべてのPTEを更新する**

ことが必要です。しかし，この方法では仮想記憶システムのコストをさらに高いものにします。

Windows 3.0では，仮想デバイスドライバ〝PHARLAP.386〟を使用するか，realcopy()関数を使用して実現されます。

回そのほか

次に述べる機能は，386|DOS-Extenderに依存していませんので，DPMI Ver.0.9でも問題は生じません。しかし，DPMI Ver.1.0のオプション機能にはどういうものがあるのか知っておくのもおもしろいでしょう。

DPMI Ver.1.0のファンクション0401Hによって，DPMI Ver.1.0でオプション扱いになっている機能を，そのホストでサポートしているかどうかを知ることができます。報告されるのは，

- PTEのAccessed，Dirtyビットの読み書き(ファンクション0506H，0507H)
- DPMIホスト内の例外の再スタート可能
- デバイスマッピング(ファンクション0508H)
- コンベンショナルメモリマッピング(ファンクション0509H)
- ページ割り当て時の0初期化
- 特権レベルクライアントの書き込み保護
- DPMIホストの書き込み保護

などです。「DPMIホスト内の例外の再スタート」能力が有効であると，DPMIホスト内で発生したフォールトを起こした命令の例外を，DPMIクライアントが例外ハンドラで処理することにより，DPMIホストは再び命令を再会できます。この能力によって，仮想記憶をもたないDPMIホストでも，クライアントが仮想記憶をサポートできます。

通常のDOS環境で実行されると、[illegible]

ルメモリ[illegible]

すると、リアルモード時には1Mバイト内に見えるセグメント[illegible]

[illegible]

ると、プロテクトモードからリアルモードコードのデータにアクセスできま[illegible]

このした"−REALBREAK"スイッチによって、最終的にコンベンショナル[illegible]

して参照できるようにすることは、DPMI Ver.0.9の範囲ではサポートされま

せん。そのため、仮想86モードでDOSに常駐されている論理[illegible]

バイト境界以下にマップされている物理メモリを、32ビット[illegible]

[illegible] PTE [illegible]

するためのものを提供すること[illegible]

DPMI Ver.1.0では、ファンクション0508Hでコンベンショナルメモリ[illegible]

[illegible]

コンベンショナルメモリ（プロテクトモードコード[illegible]

[illegible]

可能にしておきます。ファンクション0509Hは、リアルモード[illegible]

間実行の1Mバイト[illegible]

DPMI Ver.1.0でもオプション[illegible]

Windows [illegible]

ンクション0508HのようなPTEエイリア[illegible]

次のような理由によります。[illegible]

では、そのひとつのPTEの操作がほかの[illegible]

場合があるからです。これを暗黙的なPTEへの操作である[illegible]

[illegible]

する際に。

第4章

Microsoft Windows

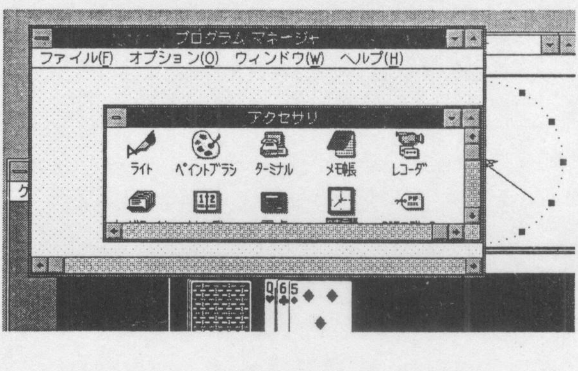
1 Microsoft Windowsとは

1.1 新しい動作環境

Microsoft Windowsは，MS-DOSにウィンドウ環境とマルチタスク機能を付加する動作環境です。MS-Windows自身は単独では動作せず，MS-DOSにかぶさるようにして動作するのでOSとはいえませんが，外見上はMS-DOSとはまったく違うOSのように見えます(fig.4.1)。

Windowsはグラフィックスやマルチタスク機能をサポートしているため，非常に大きなCPUパワーを必要とします。当初，8086で動作するようにつくられましたが，実用的になるのは，やはり386/386SXからでしょう。

● fig.4.1 ● MS-Windows Ver.3.0の画面



1.2 ユーザインターフェイス

元来，MS-DOS はキャラクタベースであり，次のような，実にそっけないユーザインターフェイスしかもっていませんでした。

A>

はじめてこの MS-DOS のプロンプトを見たユーザは，自分がこれから何をすべきかを理解することはまず不可能でした。それに対して MS-Windows は，fig.4.1 のように，ウィンドウを使って視覚に訴える，いわゆるビジュアルインターフェイスをもっています。MS-Windows であれば，たとえパソコンに詳しくなくとも，表示されたアイコンなどを見るだけでパソコンを使っていくことができるでしょう。

ビジュアルインターフェイスの基礎としては，1970 年代，米 XEROX 社の Palo Alto 研究センターで開発され，同社の Alto，Star などに採用されたものがあります。その後，このビジュアルインターフェイスは Apple 社の Lisa に採用され，さらに Macintosh に搭載されて，世界中に広く知られるようになりました。1984 年 1 月のことです。

MS-DOS 環境に，このようなビジュアルインターフェイスを付与するのが MS-Windows というわけです。

1.3 Windows の歴史

Windows の最初の版（Ver.1.01）が発売されたのは，1985 年 11 月のことでした。そのウィンドウは，

タイルドウィンドウ（Tiled Window）

と呼ばれるものです。これは画面を分割して使用するウィンドウでした。したがって，ウィンドウ同士が重なり合うことはありませんでした。

2 年後の 1987 年 11 月には，Ver.2.0 が発売されました。最大の変更点は，

ウィンドウがタイルドウィンドウから，

オーバーラップウィンドウ(Overlapped Window)

に変わった点です。これによりウィンドウを重ねることができるようになり，机の上に置いた書類と同じ感覚で扱えるようになりました。そのほか「メニュー」,「ダイアログボックス」などが変更されています。また，スワッピングを高速にするためにEMSがサポートされました。Windows 2.1では，さらにXMSがサポートされ，アプリケーションが使用できるメモリも増加し，386/386SX用には仮想86モードを使用したEMSドライバも付加されました。

その後，386/386SXの仮想86モードを使用して，DOSアプリケーションとWindows 2.xアプリケーションを並行動作させるWindows/386が発売されました。Windows/386以前のバージョンでも，DOSアプリケーションを動作させることはできましたが，フルスクリーン表示(画面全体をそのアプリケーションが占有して使う表示)のみでした。Windows/386では仮想マシンや仮想VRAMという考えを導入することにより，従来のDOSアプリケーションもウィンドウ環境下で実行できるように変更されました。

そして，Windowsのもっとも新しい版(Ver.3.0)が発表されました。1990年5月22日のことです。それまでのシェルは大幅に改良されました。**デマンドページング**による仮想記憶がサポートがされており，アプリケーションに対するメモリの制約が排除されました。これらWindowsの流れを一覧すると，次の表のようになります。

◉ **Windowsの流れ**

バージョン	IBM-PC	PC-9801
Windows 1.0	1985年 7月	1987年 6月
Windows 2.0	1987年 9月	1988年 9月
Windows/386	1987年11月	1988年10月
Windows 2.1	1988年 6月	1989年 6月
Windows 3.0	1989年末	1991年 2月

1.4 メモリと Windows 3.0

Windows は，メモリを大量に消費することが常に問題になっていました。

Windows のもとでは複数のアプリケーションを動かすことができましたが，Windows のシステム自体がメモリのかなりの部分を占めてしまい，アプリケーションが使えるメモリは限られたものでした。その限られたメモリに，複数のアプリケーションをロードして実行するわけですからたいへんです。**メモリコンパクション**や，メインメモリより大きなアプリケーションを実行する機能をもってはいますが，ディスクとのスワッピングに時間がかかるなど，不満がありました。

Windows 2.x では，EMS や XMS がサポートされたためにスワッピングは高速になり，アプリケーションが使用できるメモリも増加しましたが，決して十分とはいえませんでした。

Windows 3.0 では，386/386SX のデマンドページング機能とプロテクトモードを利用することで，メモリ管理機能が大幅に改善されました。386 エンハンストモードの場合には，最大 4G バイトもの**仮想記憶**を使うことができるようになったのです。ここにきて Windows は，はじめて“使える”ウィンドウシステムになった，といえるでしょう。メモリの制約に対する事実上の撤廃，非常に美しくユーザフレンドリーな**ビジュアルインターフェイス**など，これからのパソコンの動作環境として，とてもすぐれたものになっています。Windows 3.0 の登場により，Windows はますます普及していくものと考えられます。

この Windows 3.0 は 80286 でも動作しますが，386/386SX で実行してこそ意味があります。Windows 3.0 には，**リアルモード**，**スタンダードモード**，**386 エンハンストモード**の 3 つの動作モードがありますが，4G バイトの仮想メモリ空間は 386 エンハンストモードでしか利用できません。そして，このモードは 386/386SX を搭載したシステムでないと使えません。386/386SX の特徴である**プロテクトモード**，**デマンドページング**，**仮想 86 モード**なくして Windows 3.0 は成り立たないといえるでしょう。

次節以降では，Windows 2.x と Windows/386，Windows 3.0 の各機能を比較検討していくことによって，386/386SX とのかかわりについて見ていこうと思います。

2 MS-Windows 2.x

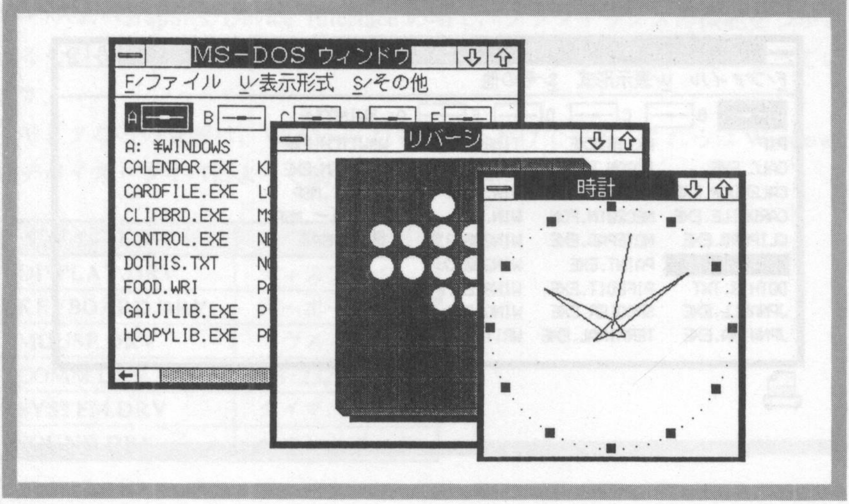
Windows 2.x は，1.x から以下の点が拡張されたものです。

- **●オーバーラップ方式のウィンドウ表示(fig.4.2)**
- **●拡張メモリのサポート**
- **●パフォーマンスの向上**
- **● DOS アプリケーションでもカットアンドペースト可能**

Windows 1.x のアプリケーションはそのまま使用できます。そして，ユーザインターフェイスは当時の OS/2 と同様になりました(Ver.1.x では，アプリケーションインターフェイスが違うため，Windows アプリケーションを OS/2 で即実行するというわけにはいきません)。

Windows 2.x は，8086 以上の CPU を搭載した DOS パソコンで動作します。

● fig.4.2 ●オーバーラップ方式のウィンドウ表示



PC-9801 で Windows 2.1 を動作させるために必要な環境は，次の通りです。

◆機種

PC-9800 シリーズ（PC-98LT を除く）

◆メモリ容量

ノーマルモード	640K バイト
ハイレゾリューションモード	768K バイト

◆必要な OS

日本語 MS-DOS Ver.3.1 以上

◆ディスク装置

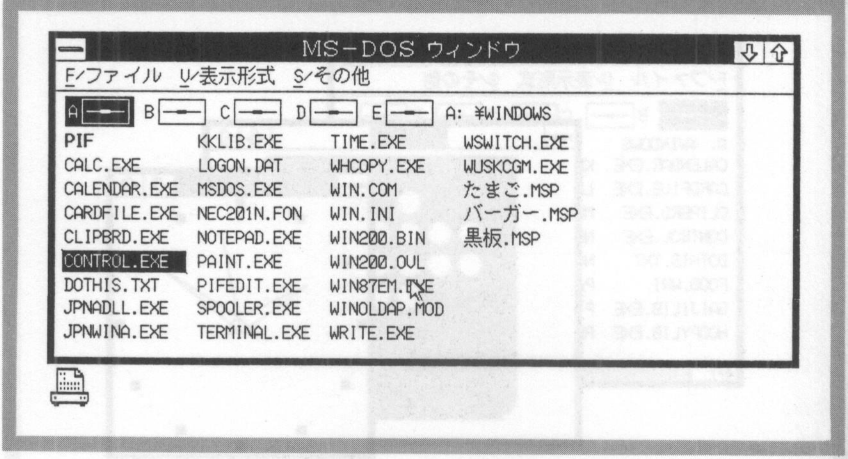
1M バイトタイプフロッピーディスクユニット 2 台以上
またはハードディスク 10M バイト以上

PC-9801 版 Windows 2.1 は，

A＞WIN ⏎

で起動できます（fig.4.3）。

● fig.4.3 ● Windows 2.x の画面



2.1 Windows 2.xの構造

WindowsはDOS上で起動します。そのためDOSアプリケーションのひとつのようにも見えますし，外見上はOSのようにも見えます。Windowsは，DOSにウィンドウ，マルチタスクなどの機能を追加し，ひとつのOSをつくりだしているといえるでしょう。

Windowsが起動すると，DOSは主としてファイルの管理を行います。そのほかのハードウェア資源であるモニタ，キーボード，マウス，RS-232C，プリンタなどは，Windowsが管理します。**メモリ管理**，タスクの**スケジューリング**も(もともとDOSにはこの機能はありませんが)，Windowsが管理しています。

Windowsは，次の3つのモジュールに分かれています。すなわち，

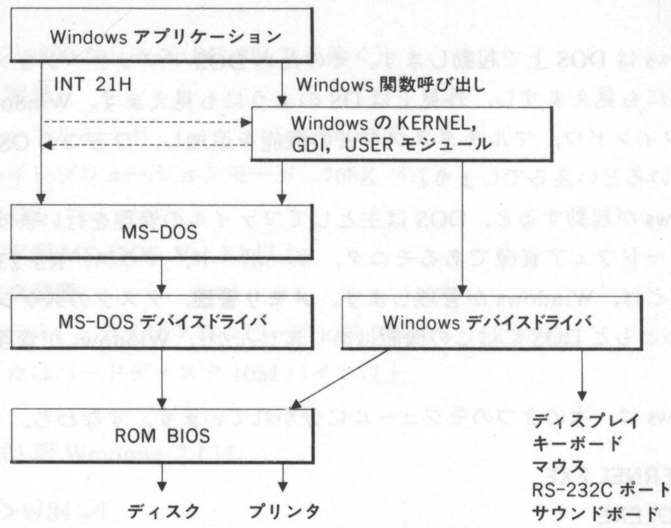
- KERNEL.EXE
- GDI.EXE
- USER.EXE

です。KERNELモジュールはプログラムの実行，スケジューリング，メモリ管理を行っています。Windowsのまさに核となるモジュールです。GDIモジュールは，Graphics Device Interfaceの略で，グラフィックスの機能をつかさどるモジュールです。USERモジュールは，それ以外の機能を受けもっています。

モニタなどの資源は，Windowsが管理するわけですが，これらはWindowsのデバイスドライバによって制御されています。

デバイスドライバ	機　能
DISPLAY.DRV	ディスプレイ
KEYBOARD.DRV	キーボード
MOUSE.DRV	マウス
COMM.DRV	RS-232C
SYSTEM.DRV	タイマ
SOUND.DRV	サウンドボード

● fig.4.4 ● Windows 2.xの全体像



これらのデバイスドライバは，GDI モジュールとUSER モジュールから呼び出され，デバイスドライバが実際のハードウェア(またはROM BIOS)を呼び出すわけです。この Windows の構造と Windows アプリケーション，ハードウェアとの関係を，fig.4.4 に示します。

図中で，Windows が内部割り込み INT 21H(DOS のファンクションコール)を横取りしていることに注意してください。これは，たとえばアプリケーションが DOS のコンソール出力を行おうとしていることを考えれば理由がわかると思います。DOS が画面に文字を出力したのでは，Windows の画面との表示が混ぜ合わさってしまいます。そのため，Windows が INT 21H をトラップして，コンソール出力をウィンドウ環境での表示に置き換えているわけです。

2.2 メモリ管理

Windows は DOS の上にかぶさるようにして動作します。そのため，アプリケーションが使用できるメモリは，DOS 単体の場合よりもさらにひっ迫します。Windows では，複数のアプリケーションをロードして実行することができますが，その場合のメモリ量はいっそう深刻です。そこで，Windows ではメインメモリが足りなくなった場合，ディスクなどにアプリケーションや Windows システムの一部を**スワッピング**します。スワッピングが頻繁に起こると動作は遅くなりますが，Windows 2.x では，EMS(p.195 を参照)をスワッピング領域に利用することもできます。こういう機構によって，メインメモリよりも大きなプログラムを実行することも可能です。そのために Windows では，メモリ管理にいろいろな工夫がなされています。

2.3 Windows 2.x のマルチタスク

マルチタスクというと，普通はシステムクロックにもとづいてタスクを管理し，スケジューリングすることが思い浮かびます。これを

プリエンプティブなマルチタスク(preemptive multitasking)

と呼びます。ところが，Windows 2.x はこのようなマルチタスクを行っていません。それは次のような方法を採用しているからです。

Windows には，メッセージをメッセージキューから取り出すための関数 GetMessage があります。アプリケーションが GetMessage 関数を呼び出したとき，メッセージキューにデータが入っているならば，処理はそのままですが，まだ入っていない場合，Windows はタスクを切り替えます。メッセージ入力では，待たされることが多いので，この処理は妥当だと思います。そこで Windows で行われているこのようなマルチタスクのことを

ノンプリエンプティブなマルチタスク(nonpreemptive multitasking)

と呼びます。実際には，メッセージが入っていても，WM_PAINT メッセージや WM_TIMER メッセージといった優先順位が低いものしか入っていない場合でも，タスクは変わります。また，GetMessage 関数だけでなく，PeekMessage 関数，WaitMessage 関数が呼び出されたときにも，タスクが変わる可能性があります。

Windows のマルチタスクは，このようにノンプリエンプティブであるため，**疑似マルチタスク**などと評されるわけです。

つまり，もしアプリケーションが Windows に処理を返さなければ，タスクは永久に変わらないのです。

Windows 2.x は，このように

オーバーラップ方式のウィンドウ
拡張メモリのサポート

などの拡張がなされ，機能としては魅力的なものでしたが，アプリケーション用のメモリは依然として不足がちでした。この状況は，Windows 3.0 が登場するまで続きます。

3 MS-Windows/386

Windows/386 は，Windows 2.1 を 386/386SX 用に拡張したもので，複数のアプリケーションを同時実行するための本格的なマルチタスク機能を備えています。Windows/386 は，外見上は Windows 2.x とほとんど変わりません。もちろん，DOS アプリケーションと Windows アプリケーションを同時に実行できます。

しかし，Windows/386 は，386/386SX の特徴である**仮想 86 モード**を利用して，DOS アプリケーションを同時に複数個並列実行できるようになっています。従来，DOS アプリケーションはフルスクリーンでしか実行できませんでしたが，Windows/386 ではウィンドウ環境下で実行可能となりました。

Windows/386 を動作させるには，当然のことながら 386/386SX を搭載したパソコンでなければなりません。PC-9801 で Windows/386 を実行するために必要な環境は，次の通りです。

◆機種

386/386SX を搭載した PC-9800 シリーズ

◆メモリ容量

ノーマルモード	最低 3.6M バイト
ハイレゾリューションモード	最低 4.5M バイト

◆必要な OS

日本語 MS-DOS Ver.3.1 以上

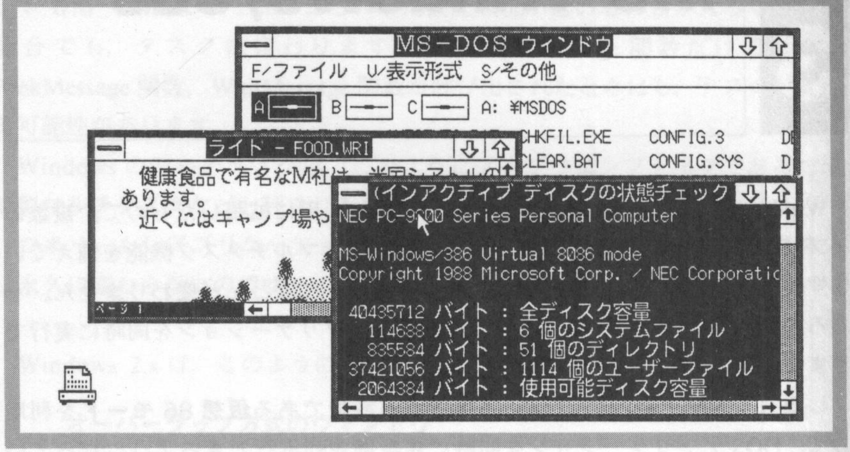
◆ディスク装置

1M バイトタイプフロッピーディスクユニット 2 台以上

ハードディスク 10M バイト以上

PC-9801 版 Windows/386 は，

● fig.4.5 ● Windows/386の画面



A>WIN386 ↵

で起動できます(fig.4.5)。

3.1 仮想マシン

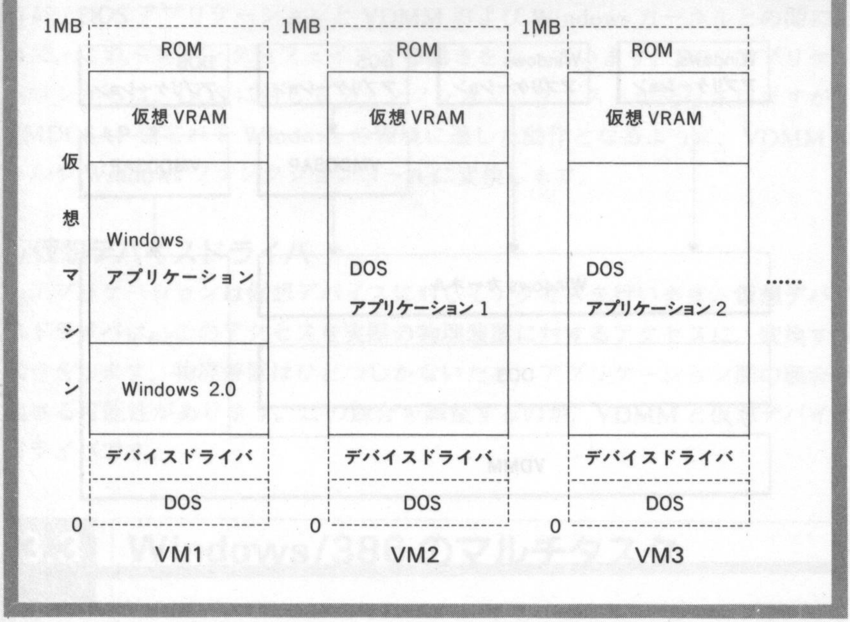
Windows/386は，386/386SXの大きな特徴である仮想86モードを使い，1台のパソコン上に何台ものパソコンがあるような状態をつくりだしています。それぞれの，いわば仮想的なコンピュータのことを

仮想マシン(VM：Virtual Machine)

と呼びます。WindowsアプリケーションやDOSアプリケーションは，それぞれ仮想マシン上で動作します。この中でWindowsが動作している仮想マシンのことを，とくにVM1と呼びます(fig.4.6)。

複数のマシンがあるように見せるといっても，実際にはひとつのハードウェアしかありません。たとえば，文字やグラフィックスを表示するVRAMもひとつです。そこで，仮想VRAMというものを用意し，そこに対するアクセス

● fig.4.6 ●仮想マシンの概念



をウィンドウに対するアクセスに変換することにより，従来のDOSアプリケーションもウィンドウの中で実行できるようになりました。Windows 2.x では，ハードウェアを直接アクセスするアプリケーションは，フルスクリーンでしか動作しませんでしたが，Windows/386 ではウィンドウ環境下でも動作します。

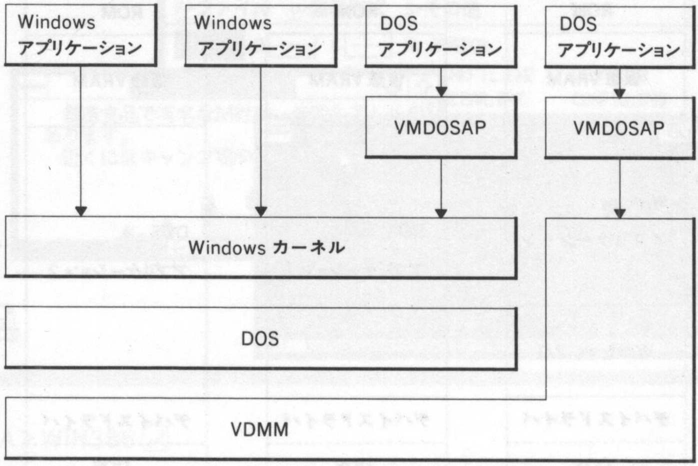
3.2 内部構成

Windows/386 の内部構成は，fig.4.7 のようになっています。

主な構成要素は，次の通りです。

●仮想 DOS マシンモニタ

● fig.4.7 ● Windows/386の内部構成



● VMDOSAP(アプリケーションとのインターフェイス)
●仮想デバイスドライバ

□仮想DOSマシンモニタ

仮想DOSマシンモニタは，VDMM(Virtual DOS Machine Monitor)と呼ばれます。Windows/386では，DOSアプリケーションを起動するごとに仮想DOSマシンを起動します。これはDOSを含んだもので，VDMMはこれらの仮想DOSマシンのスケジューリングを行います。DOSはシングルタスクOSなので，ファイルの共有とか排他制御の機能をもっていません。ところが，Windows/386はマルチタスクですから，各仮想マシンが同じファイルをアクセスすることもありえます。そこで，VDMMはファイルの共有/排他制御に関する機能ももっているわけです。

□ VMDOSAP

VMDOSAP（Virtual Machine DOS Application）は，fig.4.7からもわかるように，DOSアプリケーションとVDMMおよびWindowsカーネルとの間にあって，これらをインターフェイスする働きをもっています。DOSアプリケーションからは，DOSに対してファンクションリクエストが発行されますが，VMDOSAPはこれをWindowsの環境に適した動作となるように，VDMMコールやWindowsファンクションコールに変換します。

□仮想デバイスドライバ

アプリケーションは仮想デバイスに対してアクセスを行います。**仮想デバイスドライバ**は，このアクセスを実際の物理装置に対するアクセスに，変換する働きをします。物理装置はひとつしかないため，アプリケーション間の競合が起きる可能性があります。この競合を調整するのが，VDMMと仮想デバイスドライバです。

3.3 Windows/386のマルチタスク

Windows/386は複数の仮想DOSマシンが存在すると述べましたが，スケジューリングもこの「仮想DOSマシン単位」に行われています。Windows 2.xまでは，アプリケーションがGetMessage関数などを実行しないかぎり，タスクが切り替わりませんでした。いわゆるノンプリエンプティブなマルチタスクだったわけですが，Windows/386では，ある仮想DOSマシンが一定の時間，動作したならば，ほかの仮想DOSマシンに制御が移るようになっています。つまり，プリエンプティブなマルチタスクとなっているのです。タスクの切り替えは仮想DOSマシン単位です。Windowsアプリケーションは仮想マシンVM1で実行されますが，アプリケーションが複数ある場合は，この仮想DOSマシン内で，ノンプリエンプティブなスケジューリングによるマルチタスクが行われます。

3.4 DOS アプリケーションの実行形式

DOS アプリケーションの実行形式には，次の 3 つがあります。

- **フォアグラウンドアプリケーション**
- **バックグラウンドアプリケーション**
- **排他アプリケーション**

フォアグラウンドアプリケーションは，フルスクリーン表示のとき，またはウィンドウ内表示ならばそのウィンドウがアクティブなときにのみ実行されるアプリケーションです。PIF ファイルがないときや，とくに指定のない場合，DOS アプリケーションはフォアグラウンドアプリケーションとして実行されます。

バックグラウンドアプリケーションは，ほかのアプリケーションがフルスクリーンや，アクティブウィンドウで動作しているときも動作し続けるアプリケーションです。バックグラウンドアプリケーションが動作しているときは，フォアグラウンドアプリケーションの実行速度が若干低下します。バックグラウンドアプリケーションとして適しているのは，通信ソフトなどでしょう。逆に，エディタなどはバックグラウンドアプリケーションとしては不適当です。

最後に，**排他アプリケーション**はすべての資源を占有して高速に動作するものを指します。排他アプリケーションが動作しているとき，ほかの Windows アプリケーションやバックグラウンドアプリケーションは動作を停止します。また，排他アプリケーションは常にフルスクリーンで動作します。

3.5 ネットワーク

Windows/386 は後出の Windows 3.0 と違い，ネットワークに関しては何もサポートしていません。もし，Windows/386 からネットワークのドライブなどを使用したいのであれば，Windows/386 を起動する前にネットワークのドライブを接続し，Windows/386 を終了後，切り離すようにするしかありませ

ん。したがって，Windows/386 の中からネットワークドライブを接続したり，切り離したりしてはいけません。また，そのような機能も用意されていません。

Windows/386 は，このように 386/386SX に特化した Windows です。Windows/386 では DOS アプリケーションをウィンドウで実行することができますが，速度は非常に遅いものとなってしまいます。したがって，アプリケーションを排他アプリケーションとして実行させ，Windows/386 はタスクスイッチャとして使用することが現実的だといえます。Windows/386 では DOS タスクのウィンドウ環境での並行実行が可能になりましたが，Windows 2.x 同様，Windows アプリケーションのメモリ量は依然不足したままでした。この点は Windows 3.0 の登場まで続きます。

4 MS-Windows 3.0

Windows 3.0 が登場したのは，1990 年 5 月 22 日のことです。表示，操作などは Macintosh ライクになり，CPU が 386/386SX のマシンの場合には，最大 4G バイトの仮想記憶をもち，DOS アプリケーションを複数個並列して動作させることが可能になりました。Windows 2.1 と 3.0 の違いは，次の表のようにまとめることができるでしょう。

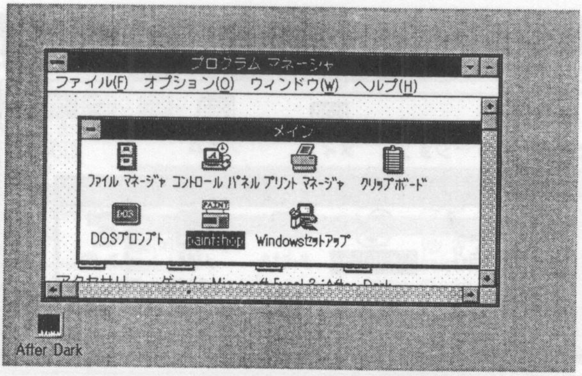
事項＼バージョン	Windows 2.1	Windows 3.0
シェル	MS-DOS エグゼクティブ	プログラムマネージャ ファイルマネージャ タスクリストの 3 種類
ファイル名	文字列で表示	アイコン/ツリー構造で表示
フォント	固定	プロポーショナル
カラービットマップ	機種に依存	機種から独立
オンラインヘルプ	なし	あり
CPU モード	リアルアドレスモード	リアルアドレスモード 80286 保護仮想アドレスモード 386 保護仮想アドレスモード ＋仮想 86 モード，以上 3 種類
メモリ管理	実メモリ	仮想メモリ
LAN	考慮せず	リモートドライブ， リモートプリンタのサポートあり

PC-9801 で Windows 3.0 を動作させるために必要な環境は，次の通りです。

◆機種

80286 以上の PC-9800 シリーズ（PC-98XA は除く）

● fig.4.8 ● Windows 3.0の画面



◆**メモリ容量**

【リアルモード時】

ノーマルモード　最低 1.6M バイト

ハイレゾリューションモード　最低 1.5M バイト

【スタンダードモード時】

ノーマルモード　最低 1.6M バイト

ハイレゾリューションモード　最低 1.5M バイト

【386 エンハンストモード時】

ノーマルモード　最低 3.6M バイト

ハイレゾリューションモード　最低 4.6M バイト

◆**必要な OS**

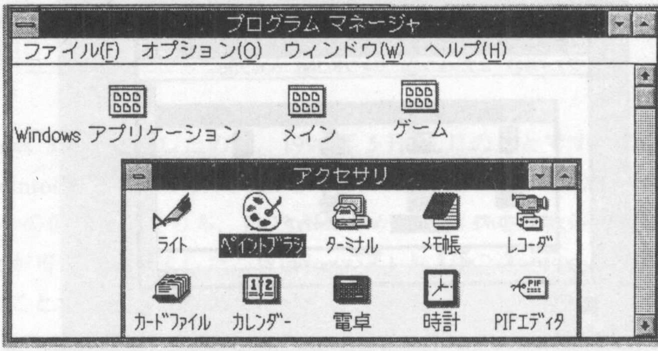
日本語 MS-DOS Ver.3.3 以上

◆**ディスク装置**

1M バイトタイプフロッピーディスクユニット 1 台以上

ハードディスク 8M バイト以上

● fig.4.9 ●プログラムマネージャ



PC-9801版 Windows 3.0は,

　　A＞WIN ⏎

で起動できます(fig.4.8)。

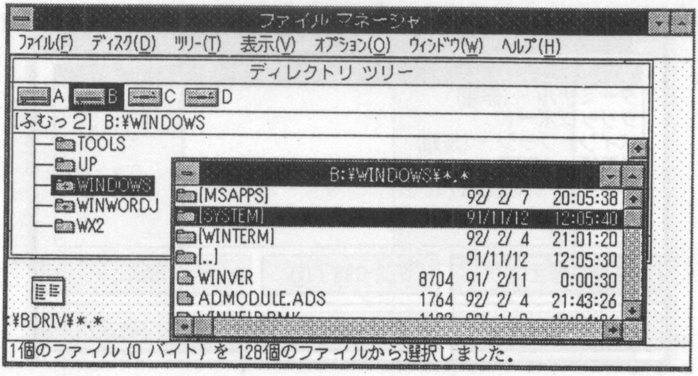
4.1 シェル

Windows 2.1にはMS-DOSエグゼクティブという，そっけないシェルがひとつだけでしたが，Windows 3.0では3種類のシェルが用意されました。すなわち

- **プログラムマネージャ**
- **ファイルマネージャ**
- **タスクリスト**

です。これらの機能はOS/2のプレゼンテーションマネージャにもあり，よく似たものとなっています。

● fig.4.10 ●ファイルマネージャ

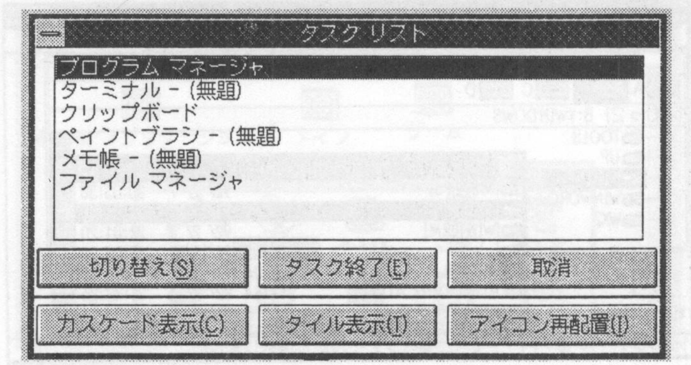


プログラムマネージャは，ファイルがアイコンで表示されます(fig.4.9)。Macintoshとよく似た画面構成となっているのがおわかりいただけるでしょう。プログラムマネージャではいくつかのソフトウェアをグループ化して登録することができます。これらのソフトウェアは同じディレクトリである必要はありません。fig.4.10のように各グループ，各ソフトウェアはそれぞれアイコンで表示され，当然ながらマウスで選択することができます。

プログラムマネージャがファイルをグループでまとめて扱う，いわば論理的なものであるのに対し，**ファイルマネージャ**は，DOSのディレクトリ単位にファイルを扱う物理的なものです。図を見ればわかるように，ファイルマネージャ中ファイルは，ツリー構造で表示されます。ここからソフトウェアを起動したり，削除，コピーといった作業を行うことができます。

最後の**タスクリスト**は，現在実行中のタスクの中からフォアグラウンドで実行するものを選んだりするためのものです(fig.4.11)。

● fig.4.11 ●タスクリスト



4.2 動作モード

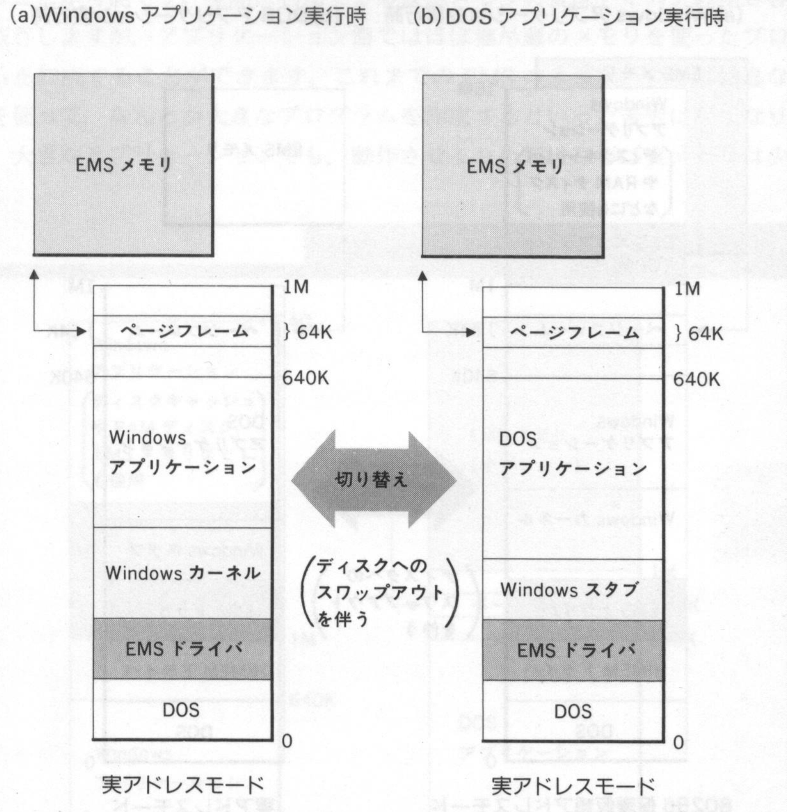
Windows 3.0 には，次の 3 つの動作モードがあります。

- **リアルモード**
- **スタンダードモード**
- **386 エンハンストモード**

回リアルモード

リアルモードは，CPU の実アドレスモードで動作し，従来の Windows アプリケーションをそのまま動作させることができます。その代わり，使用できるメモリ量も従来通りです。DOS アプリケーションも実行できますが，その場合 Windows システムはディスクに退避することになります。複数の DOS アプリケーションの並行処理も可能ですが，一度に動作するのはひとつだけで，ほかのソフトウェアは，やはりディスクに退避されます。リアルモード時のメモリマップは，fig.4.12 の通りです。

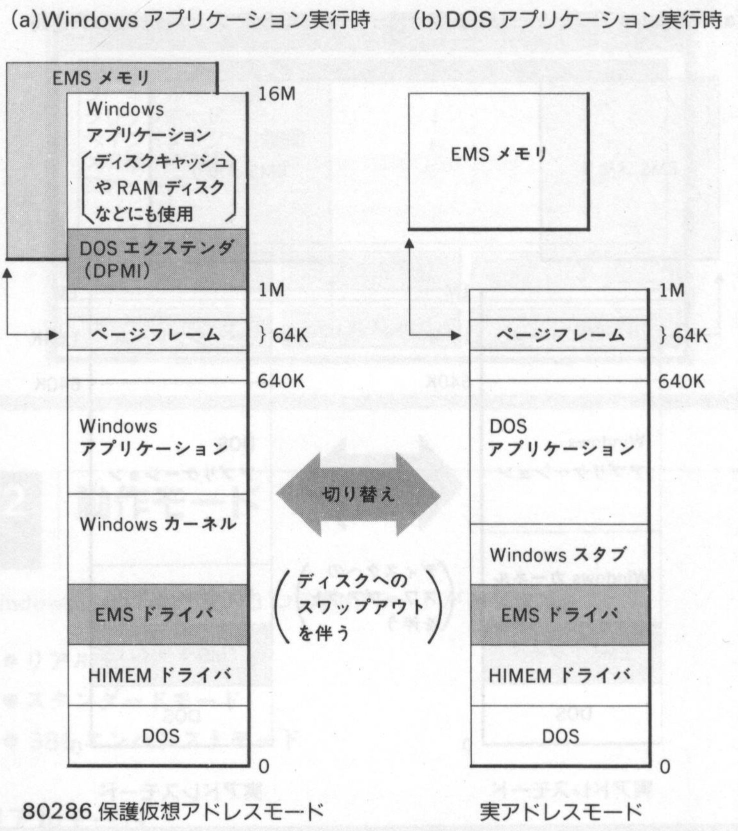
● fig.4.12 ●リアルモードのメモリマップ



◻スタンダードモード

スタンダードモードは，80286 の仮想アドレスモードで動作します。Windows アプリケーションは仮想アドレスモードで実行し，最大 16M バイトの実メモリを使うことができます。DOS アプリケーションは実アドレスモードで実行します。複数個を動作させることもできますが，動作していないアプリケーションをディスクに退避するのは，リアルモードと同様です。

● fig.4.13 ●スタンダードモードのメモリマップ



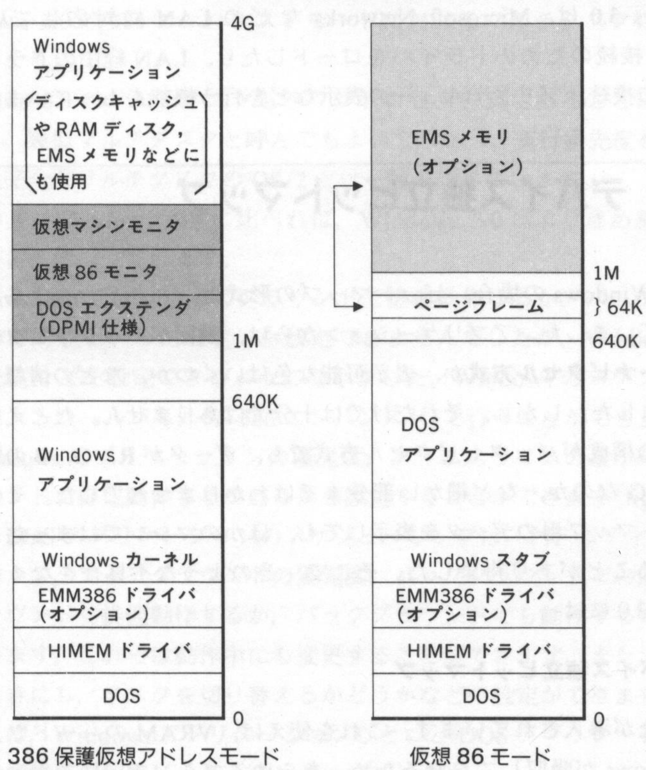
スタンダードモード時のメモリマップを，fig.4.13に示します。

□ 386エンハンストモード

386エンハンストモードは，386/386SXの保護仮想アドレスモードで動作します。386エンハンストモードこそが，386マシンの能力を最大限に使ったモードといえます。386エンハンストモードは，Windows/386の機能をさらに

推し進めています。386/386SX に用意されている仮想記憶方式である**デマンドページング方式**を使用しており，これにより最大 4G バイトの仮想メモリを使うことが可能です。実際に利用できる仮想メモリの量はディスクの空き容量に依存しますが，アプリケーション側ではほぼ無尽蔵のメモリを使ったプログラムを作成することができます。これまでの EMS のような，いわば姑息な手段を使って，なんとか大きなプログラムを作成するといった苦労はなくなります。大きなアプリケーションでも，動作させるためだけならば実メモリは少な

● fig.4.14 ● 386 エンハンストモードのメモリマップ



くてもよいのですが，実メモリが少ないとページングのためのディスクアクセスが増えて実行速度が低下するため，できるだけ多くの実メモリを搭載するほうが現実的でしょう。

従来のDOSアプリケーションは，**仮想86モード**を使って複数並行に実行することができます。

386エンハンストモード時のメモリマップは，fig.4.14(p.185)の通りです。

4.3 LANへの対応

Windows 3.0は，Microsoft NetworksなどのLANに対応しています。LANとの接続のためのドライバをロードしたり，LAN経由のドライブやプリンタとの接続，プリントキューの表示などを行う機能をもっています。

4.4 デバイス独立ビットマップ

従来のWindowsの場合，ビットマップの形式は，ハードウェアをそのまま反映させていました。アプリケーションからは，画面が**パックトピクセル方式**か，**プレーナピクセル方式**か，表示可能な色はいくつか，などの情報を知ることはできました。しかし，それだけでは十分ではありません。たとえば，ハードウェアの構成がパックトピクセル方式でも，データがR，G，Bの順番なのかB，R，Gなのか，など細かい部分まではわかりませんでした。そのため，同じビットマップ用のデータを表示しても，ほかのマシンではまったく違った表示になることがあり得ました。そこで，そのような不具合をなくすためにWindows 3.0には

デバイス独立ビットマップ

という概念が導入されています。これを使えば，VRAMのハードウェアの違いをWindowsが吸収してくれるため，あらゆるマシンで同様な表示を行うアプリケーションを作成することが可能になります。

Windows 3.0 でのデバイス独立ビットマップ関数は，次の表の通りです。

関　数	機　能
CreateDIBitmap	デバイス独立な情報をもつビットマップの作成
GetDIBits	ビットマップからデータを取得
SetDIBits	ビットマップにデータをセット
SetDIBitsToDevice	デバイス独立ビットマップの表示

4.5 Windows 3.0 のマルチタスク

Windows 3.0 の 386 エンハンストモードにおけるマルチタスクについて説明しましょう。

Windows 3.0 のマルチタスクもラウンドロビン方式であることに違いはありません。疑似マルチタスクと呼んでもよいでしょう。実行優先度を細かく設定できる完全なマルチタスクの OS/2 とは一線を画しています。

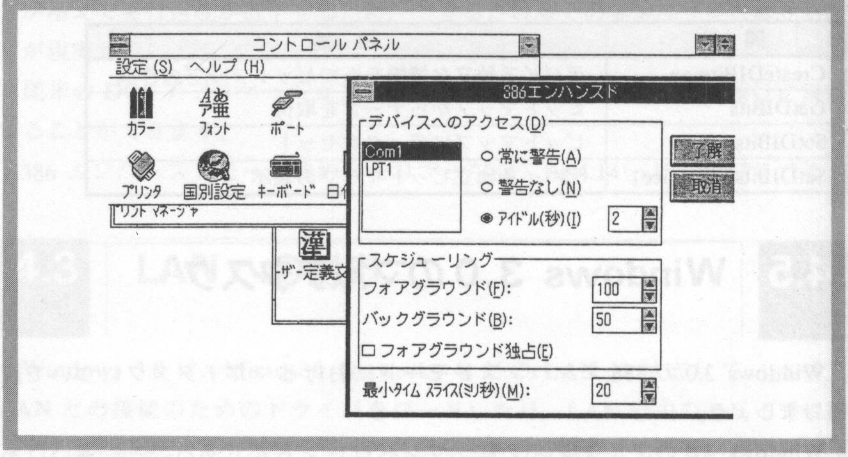
とはいえ，Windows/386 に比べれば，Windows 3.0 はよりきめ細かい制御ができるようになっています。

Windows/386 では，フォアグラウンドで排他動作するか，バックグラウンドで動作が可能といった程度しか設定できませんでしたが，Windows 3.0 では，優先度などを設定できるようになりました。Windows のスケジューリングは，コントロールパネルの「386 エンハンスド」で行うことができます。ここでは Windows アプリケーションの優先度，タイムスライスの最小時間，フォアグラウンドで排他動作するかどうかを設定することができます(fig.4.15)。

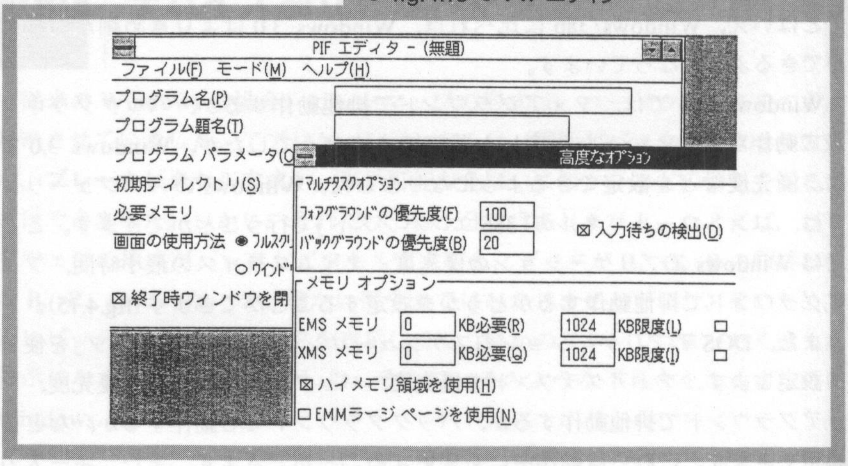
また，DOS アプリケーションのスケジューリングは，「PIF エディタ」を使って設定します。フォアグラウンドの優先度，バックグラウンドでの優先度，フォアグラウンドで排他動作するか，バックグラウンドでも動作するか，などを指定できます。これらは動作中にも変更することができます。また，キー入力待ちのときにも，タスクを切り替えるかどうかなどの設定ができます(fig.4.16)。

たとえば，Windows アプリケーションひとつと DOS アプリケーション 4 つが動作しているとしましょう。スケジューリングが，

● fig.4.15 ●コントロールパネル(386 エンハンストモード)



● fig.4.16 ● PIF エディタ



フォアグラウンド	100
バックグラウンド	50
最小タイムスライス	20〔ms〕

だったとします。そして，DOS アプリケーションの実行優先度は，PIF エディタを使ってそれぞれ 10，20，30，40 に設定されているとします。まず，Windows アプリケーションがフォアグラウンドで実行されていたならば，マルチタスクの 1 サイクルは，

(100＋10＋20＋30＋40)÷10×20

より

400〔ms〕

になります。そして，そのうち Windows アプリケーションに割り当てられるのが，

400÷(100＋10＋20＋30＋40)×100

より

200〔ms〕

と求められます。同様に，各 DOS アプリケーションに割り当てられるのは，それぞれ

400÷(100＋10＋20＋30＋40)×10＝20〔ms〕
400÷(100＋10＋20＋30＋40)×20＝40〔ms〕
400÷(100＋10＋20＋30＋40)×30＝60〔ms〕
400÷(100＋10＋20＋30＋40)×40＝80〔ms〕

となります(fig.4.17)。

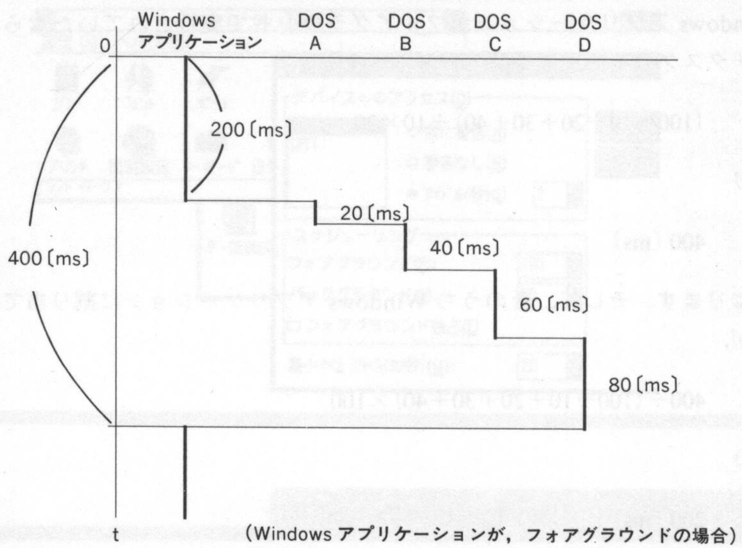
次に，Windows アプリケーションがバックグラウンドで動作している場合，マルチタスクの 1 サイクルは，

(50＋10＋20＋30＋40)÷10×20

より

300〔ms〕

● fig.4.17 ● DOS アプリケーションのスケジューリング



になります。そして，そのうち Windows アプリケーションに割り当てられるのは，

300÷（50＋10＋20＋30＋40）×50

より

100〔ms〕

です。その結果，各 DOS アプリケーションに割り当てられるのは

300÷（50＋10＋20＋30＋40）×10＝20〔ms〕
300÷（100＋10＋20＋30＋40）×20＝40〔ms〕
300÷（100＋10＋20＋30＋40）×30＝60〔ms〕
300÷（100＋10＋20＋30＋40）×40＝80〔ms〕

となります。

なお，Windows アプリケーションが複数動作している場合は，Windows 用に割り当てられた中で，さらに分割されて動作することになります。

4.6 DPMI

Windows 3.0 では，DOS アプリケーションが DOS の環境で動作するのとほぼ同じような状態をつくりだすことに注意されています。最近の DOS アプリケーションには EMS や XMS を利用するものがありますが，Windows 3.0 では，アプリケーションから EMS，XMS が利用できるようになっています。このほかに DOS **エクステンダ**を使用したアプリケーションも米国ではいくつかあるようです。たとえば，IBM-PC 版の Lotus『1-2-3』R3 などです。これは，

VCPI（Virtual Control Program Interface）

にもとづく DOS エクステンダを使用しています。Windows 3.0 には，

DPMI（Dos Protected Mode Interface）

仕様の DOS エクステンダが組み込まれています（p.115 を参照）。今後，DOS アプリケーションが DPMI 仕様の DOS エクステンダを使用するようになれば，Windows 3.0 上でも問題なく動作するようになるはずです。Windows 3.0 のスタンダードモード，386 エンハンストモードからは，DPMI 仕様の DOS エクステンダが使用可能です。

4.7 DLL

通常，ライブラリとのリンクは link エディット時に行われますが，Windows 3.0 にかぎらず Windows では，実行時にリンクすることができます。Windows アプリケーションは，ほかの〝.EXE〟ファイルとルーチンを共有す

ることができるわけです。この共有ルーチンが含まれるファイルを

DLL(Dynamic Link Library)

と呼びます。Windows 自体も DLL の寄せ集めです。従来の DOS の".EXE"ファイルでは，ダイナミックリンクについては考慮されていませんでした。そこで，Windows では DLL に対応するためにも New Executable フォーマット(p.201 を参照)を導入する必要があったわけです。".EXE"ファイルの作成には，DOS の場合，"LINK.EXE"を通常使いますが，これは New Executable フォーマットには，当然対応していません。この場合は Windows ソフトウェア開発キットに含まれる"LINK4.EXE"などを使用します。

一方，Windows では，DLL を使うことによって，プログラムのコードを小さくすることができます。これは，メモリの制約が厳しい Windows 2.x では有効な手段だったといえるでしょう。また，新しいモジュールを作成した場合などでは，単にロードするだけで古いモジュールと入れ替えることが可能になります。これはシステムを少しずつ開発していく場合などに便利です。

4.8 ヘルプマネージャ

最近は，アプリケーション自体にヘルプ機能を組み込むことが多いように見受けられます。そのため，いちいちマニュアルを参照しなくても，簡単な疑問などはアプリケーション自体が答えてくれます。しかし，このヘルプ機能はアプリケーションごとに違っており，ユーザからすれば操作の方法がそれぞれ異なる不便さがあり，プログラマ側からいえばアプリケーションごとにいちいちヘルプ機能のプログラムをつくる必要がありました。Windows 3.0 にはヘルプマネージャが搭載され，ヘルプマネージャ用にヘルプファイルを作成すれば，ヘルプの機能ができあがるようになっています。使う側からも操作性が統一されるので，好ましい機能だといえます。

4.9 付属アプリケーション

余談ですが，ここで Windows 3.0 に付属するアプリケーションについても少し触れておきましょう。

Windows 3.0 に付属している主なアプリケーションは「ペイントブラシ」，「ターミナル」，「マクロレコーダ」，「電卓」などです。「ペイントブラシ」は従来のバージョンの Windows に含まれていた「Paint」のカラー版です。「ターミナル」は通信ソフトウェアで，VT100 のエミュレーションや，Xmodem,・Kermit の機能などをもっています。「マクロレコーダ」は，マウスやキーボードの入力を記憶し再生する，いわばマウス操作を含んだバッチ機能と考えてよいでしょう。「電卓」は，従来のバージョンに，16 進機能とか科学技術計算機能などが付加されています。

それから Windows 3.0 には,「リバーシ」,「ソリティア」というふたつのゲームが付属しています。「リバーシ」は以前の版から付属するオセロタイプのゲームです。「ソリティア」はカードゲームで，1 人トランプの一種です。カードの動きがスムーズなうえ，デザインが変更できるなど，なかなかおもしろいアプリケーションでしょう。

市販の Windows 用ゲームとしては,「Microsoft Games for Windows」1/2 があります。その中で,「minesweeper」などは“はまる”ゲームといえるでしょう。Windows 用のアプリケーションは充実しているとはいえませんが，このようなゲームが普及の牽引力となってもよいのではないかと思います。

Windows は，Windows 3.0 になって初めて，このように現実的なものとなりました。とくに，386 エンハンストモードで使用した場合，386/386SX の能力を十分に引き出すことができます。

Windows 3.0 用のアプリケーションはいくつか登場していますが，DOS のアプリケーション数に比べれば，まだまだ少ないといえます。その点で Windows 3.0 は，しばらく DOS アプリケーションのスイッチャとして使われることが多いのではないでしょうか。スイッチャの機能は DOS Ver.5.0 にもありますが，Windows の機能のほうがはるかに洗練されています。

たとえばこんな具合です。プログラムマネージャの中から「メイン」をクリックし，その中からDOSプロンプトを選択します。すると，次のような通常のDOSのプロンプトが現れます。

```
NEC PC-9800 Series Personal Computer

MS-Windows 3.0 Enhanced mode DOS prompt
Copyright (C) 1990,1991 Microsoft Corp. / NEC Corporation

Command バージョン 3.30C

A>
```

この状態では，いままでのDOSと同じ使い方ができます。そして，Windowsの機能が必要になったときはいつでも GRPH ＋ TAB でWindowsを呼び出すことができます。さらに，DOSプロンプトを選択すれば，もうひとつのDOSタスクが起動します。先に起動したものとは，タスクマネージャを呼び出すことで簡単に切り替えることができます。ショートカットキーを設定すれば，簡単なキー操作だけで切り替えることも可能です。これで，たとえば一方でエディタを起動したまま，瞬時に通信ソフトに切り替えるといったことが簡単に行えます。

Windowsは，よくOS/2と比較されます。OS/2は完全なマルチタスクOSです。また，OS/2のPM(プレゼンテーションマネージャ)はWindowsと同等のビジュアルインターフェイスをもっています。OS/2があれば，Windows 3.0は必要ないように感じられますが，OS/2は完全なマルチタスクOSである半面，かなり重いプログラムとなってしまっています。エンドユーザが使用するのは，Windows 3.0のような比較的軽いプログラムのほうが適しているのではないでしょうか。Windows 3.0とOS/2は，そのあたりを基準に，今後棲み分けがなされていくと思います。

5 Windowsにみるそのほかの特徴

5.1 拡張メモリ

Windowsの歴史は，メモリ獲得の歴史ともいえます。ウィンドウ環境やマルチタスク，DLLやDDE(後述)などすばらしい機能をもっていますが，実際のアプリケーションがメモリ不足のために動かないのでは，まったく意味がありません。この葛藤はWindows 3.0になってようやく解決しましたが，Windows 2.xでもアプリケーションが使えるメモリ量を増やす方法がとられていました。EMSとXMSです。EMSはWindows 2.0から，XMSはWindows 2.1からサポートされました。EMSもXMSも，Windows専用の規格ではありませんが，Windows 2.xを快適に動作するために，ぜひ使用したい機能です。

◻ EMS

EMSはExpanded Memory Specificationの略で，1Mバイト以上のメモリをアクセスすることを可能にしたものです。EMSをパソコン上に実現するにはEMS機能をもったメモリボードを装着するか，386/386SXをもった機種でソフトウェアによりEMS機能を実現する方法があります。EMSをパソコン上で実現するソフトウェアのことを

EMM(Expanded Memory Manager)

と呼びます(p.102を参照)。これはデバイスドライバで供給されます。

PC-9801用MS-DOS Ver.3.3以上には，〝EMSDRIVE.SYS〟というEMMが付属しています。

PC-9801用Windows 2.1には〝WINEMM.SYS〟というWindows用の

EMM が付属しています。これは EMS メモリボード専用です。

本書ともっとも関係があるのは，386/386SX の仮想 86 モードを使ってソフトウェアで EMS を実現する方法でしょう。

Windows 2.1 には仮想 86 モードを使用したデバイスドライバが付属しています。〝EMM386.SYS〟です。これは 386/386SX の仮想 86 モードのページング機能を利用した EMS ドライバです。仮想 86 モードのページング機能を使って，アドレスが 1M バイトを超えた領域にあるメモリを 1M バイト以下の領域に写像することによって EMS 機能を実現しています。1M バイト以上の領域にメモリを増設しておく必要はありますが，EMS メモリボードなどを新たに購入する必要もなく，ソフトウェアを付加するだけで EMS 機能が得られるものです。

Windows 2.x は EMS が利用できる場合，メインメモリに空きがなくなると，最近アクセスされていないセグメントを EMS にスワップするようになります。すると，スワッピングが高速になり，大きなアプリケーションの実行速度が向上します。

回 XMS

XMS は，eXtended Memory Specification の略で，リアルモードから使えるメモリを増加させるための規格です。XMS を実現するソフトウェアを

XMM(eXtended Memory Manager)

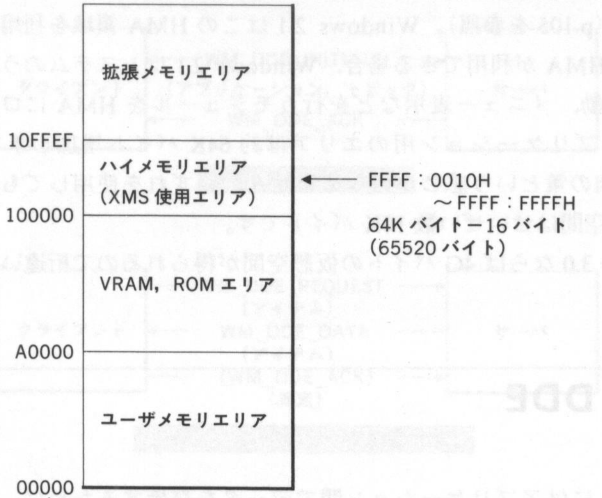
と呼びます。XMM もデバイスドライバの形で実現されています。PC-9801 用 Windows 2.1 には，〝HIMEM.SYS〟というデバイスドライバが付属しています。XMS は，80286/386/386SX でアドレスが 1M バイト以上の領域にメモリが拡張されている場合に使用できます。XMS を使用したときのメモリマップは，fig.4.18 のようになります。

8086 は，20 ビットを使ってメモリをアドレスしていました。16 進数で表すと，次の通りです。

00000H 〜 FFFFFH

もし，セグメント値が FFFFH であったならば，オフセット値に意味がある

● fig.4.18 ● Windows 2.1のXMS使用時のメモリマップ



のは，

0000H～000FH

の範囲だけでした。XMSでは，セグメント値がFFFFHでも，オフセット値はFFFFHまで取れるように工夫されています。8086ならばラップアラウンドを起こして低いアドレスを指すことになりますが，80286/386/386SXなどで1Mバイト以上のメモリを搭載している場合，高いほうのアドレス，すなわち

10000H～10FFEFH

を参照するようになっているのです。非常に巧妙な手段ですが，これによってリアルモードで動作しているDOSからアクセスできるメモリ空間が，約64Kバイト増えます。

このFFFF:10～FFFF:FFFFまでの空間のことを

HMA(High Memory Area)

と呼びます(p.105を参照)。Windows 2.1はこのHMA領域を利用することができます。HMAが利用できる場合，Windows 2.1はシステムのうちウィンドウ描画，移動，メニュー表示などを行うモジュールをHMAにロードします。そのためアプリケーション用のエリアは約64Kバイト増加します。EMS，XMSも苦肉の策という感じがいなめません。いずれを使用しても，リニアに利用できる空間はせいぜい数10Kバイトです。

Windows 3.0ならば4Gバイトの仮想空間が得られるので桁違いの環境となります。

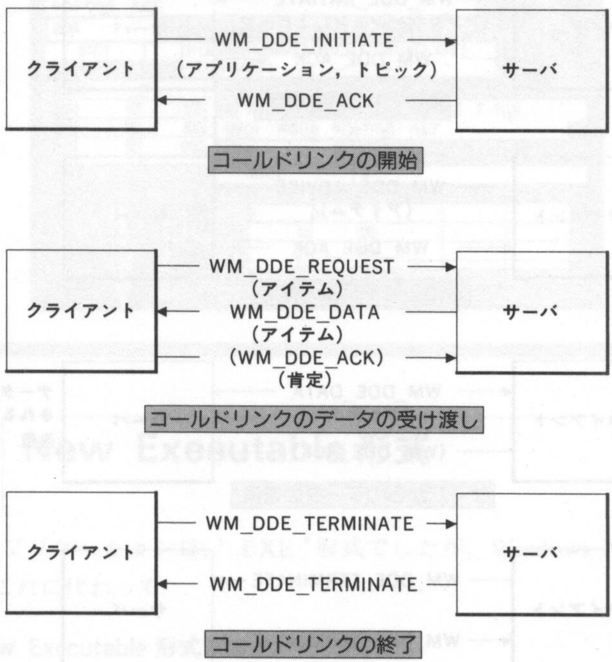
5.2 DDE

Windowsにはアプリケーション間でデータを交換するために，ふたつの方法が用意されています。ひとつは**クリップボード**による方法，もうひとつがDDEによる方法です。「クリップボード」は，メモリブロックをアプリケーションの間で交換するための一連の機能のことをいいます(Windowsに付属する〝CLIPBRD.EXE〟とは，別のものであることに注意してください)。

◉ DDEを使って交換される代表的なデータの例

名　称	内　容
CF_TEXT	NULL文字で終了する文字列
CF_METAFILEPICT	GDI(Graphics Device Interface)コマンドで表現した図形
CF_BITMAP	ビットマップ
CF_SYLK	SYLK形式
CF_DIF	DIF(Data Interchange Format)
CF_PTF	書式情報付き文字列(リッチテキストフォーマット)
CF_WK1	Lotus『1-2-3』のファイルフォーマット
CF_BIFF	『Excel』のファイルフォーマット
CF_CSV	カンマで区切ったデータ列

● fig.4.19 ● クライアントが要求したときにデータを送受する方法(コールドリンク)



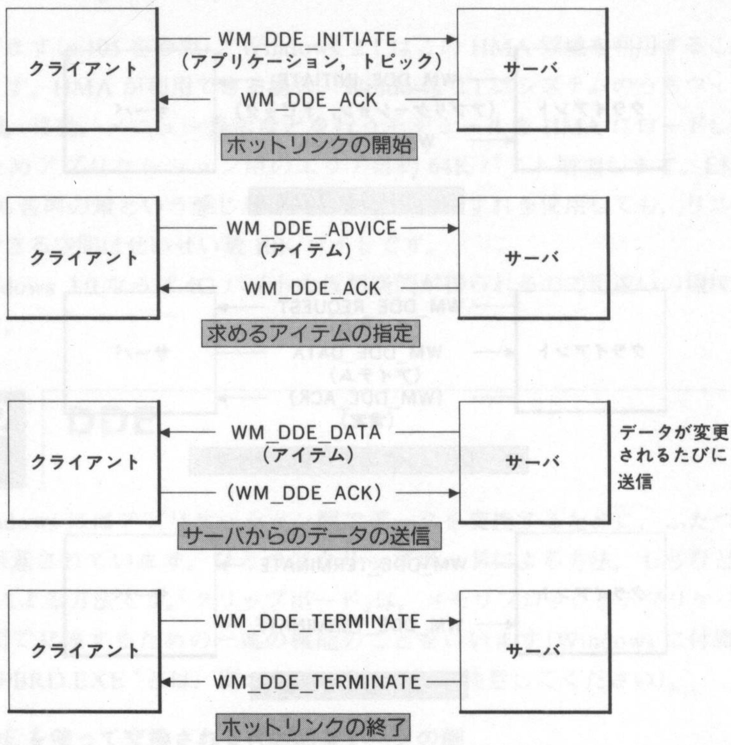
DDEは

Dynamic Data Exchange

の略で，アプリケーション間の通信プロトコルです。DDEを使うと，アプリケーション間でデータを交換することができます。

DDEは，いわゆるサーバクライアントモデルを使って通信を行います。クライアントが要求したときにデータを送受する方法(fig.4.19)と，サーバのデ

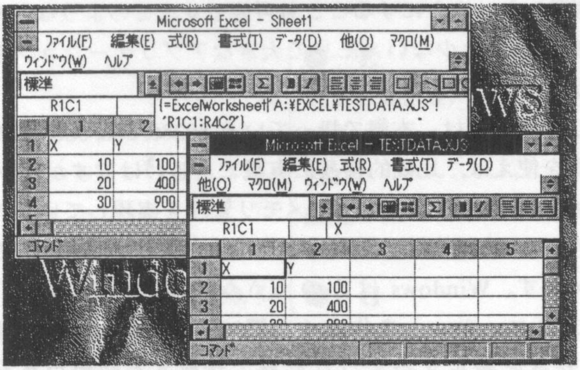
● fig.4.20 ●サーバのデータが変更されたときにクライアントに送受する方法(ホットリンク)



ータが変更されたときにクライアントに送信する方法が用意されています(fig.4.20)。

DDEを使えば高度なプログラムが開発できますが，DDEはアプリケーション側でサポートしていないと使えない機能です。『Microsoft Excel for Windows』はDDEに対応しています(fig.4.21)。

● fig.4.21 ● 『Microsoft Excel』の画面



5.3 New Executable 形式

DOSアプリケーションは，".EXE"形式でしたが，Windowsアプリケーションは，これに代わって

New Executable 形式

が使われています。New Executable 形式のファイルには，コードセグメント，データセグメントについての情報(属性)が書かれています。コードセグメントに関しては「移動可能か」,「破棄可能か」，データセグメントに関しては「移動可能か」などの属性が付けられています。これらの属性を参照することにより，もし，メモリ空間が足りなくなった場合は，不用なコードセグメントをメモリから破棄して，新しいアプリケーションをロードすることが可能になります。破棄したコードセグメントが必要になったときは，2次記憶上の".EXE"ファイルから，もう一度ロードすればよいわけです。また，ロードと破棄が繰り返されてメモリが虫食い状態になっても，移動可能な属性のセグメントならば，それらを移動してメモリコンパクションを行い，連続した大きな領域をつくっ

たりもします。コードセグメントにLOADONCALL属性，すなわち呼ばれたときに初めてメモリにロードされるという属性を付ければ，呼ばれないかぎりメモリを消費しないようにすることも可能です。このようなメモリ管理機構によって，Windowsは少ないメモリで大きなアプリケーションを動かすことができるようになっています。

このメモリ管理機構は，本書で扱っている80286/386/386SXのセグメンテーション機能を使えば，比較的容易に実現できるわけですが，Windowsでは8086のリアルモードからこのようなメモリ管理を実現しているのです。8086のアーキテクチャからすれば，これはとてつもなくたいへんなことであるとわかると思います。Windowsはこのような特殊なことをしているため，Windowsアプリケーションを作成する際には，注意が必要です。それは，Windowsの要求を満足するような形式で作成する必要があるからです。Windowsアプリケーションを開発するためには，専用の開発ツールキットが用意されています。それにはWindowsのメモリ管理などに対する要求を満たすコンパイラと，New Executable形式のモジュールを作成できるリンカなどが含まれています。

索　引

●—ま

●—ら

●—わ

●—数字

図一覧

【参考文献】
"i486(TM) MICROPROCESSOR",
Intel Corporation, 1989 年
"DOS PROTECTED MODE INTERFACE (DPMI) SPECIFICATION" Version 0.9,
The DPMI Committee, 1990 年
"DOS PROTECTED MODE INTERFACE (DPMI) SPECIFICATION" Version 1.0,
The DPMI Committee, 1991 年
"Virtual Control Program Interface" Version 1.0,
Phar Lap Software, Inc. Quarterdeck Office Systems, 1989 年
"High C(R) Programmer's Guide Version 1.6 for MS-DOS/386",
MetaWare(R) Incorporated, 1990 年

『80386 プログラマーズ・リファレンス・マニュアル』, インテルジャパン株式会社, 1987 年
『386|TOOLBOX ユーザーズマニュアル』, テックソフトアンドサービス株式会社, 1992 年
『日本語 MS-Windows/386(Ver.2.11) ユーザーズマニュアル』, NEC Corporation, 1989 年
『MS-Windows 2.11 ユーザーズガイド』, NEC Corporation, 1989 年
『MS-Windows 3.0 ユーザーズリファレンスマニュアル』, NEC Corporation, 1991 年
『80386 プログラミング』, 工学社, 1988 年
『ザ 80386 ブック』, 富士ソフトウエア, 1989 年
『プログラミング WINDOWS VOLUME1』, アスキー出版局, 1989 年
『プログラミング WINDOWS Version3』, アスキー出版局, 1991 年
『日経バイト』, 日経 BP 社, 1988 年 12 月, 1990 年 7 月, 1990 年 8 月

MS-DOSユーザーズ386パソコン活用ブック

定価はカバーに記載
されております

1992年5月22日 初版印刷
1992年6月1日 初版発行

著者 根津半蔵/星野進二
発行者 孫 正義
発行所 ソフトバンク株式会社
出版事業部
〒108 東京都港区高輪2-19-13 NS高輪ビル
営業☎03(5488)1360 編集☎03(5488)1326
制作/印刷 有限会社バリエ社

Printed in Japan ISBN 4-89052-303-0

落丁, 乱丁はお取り替えします

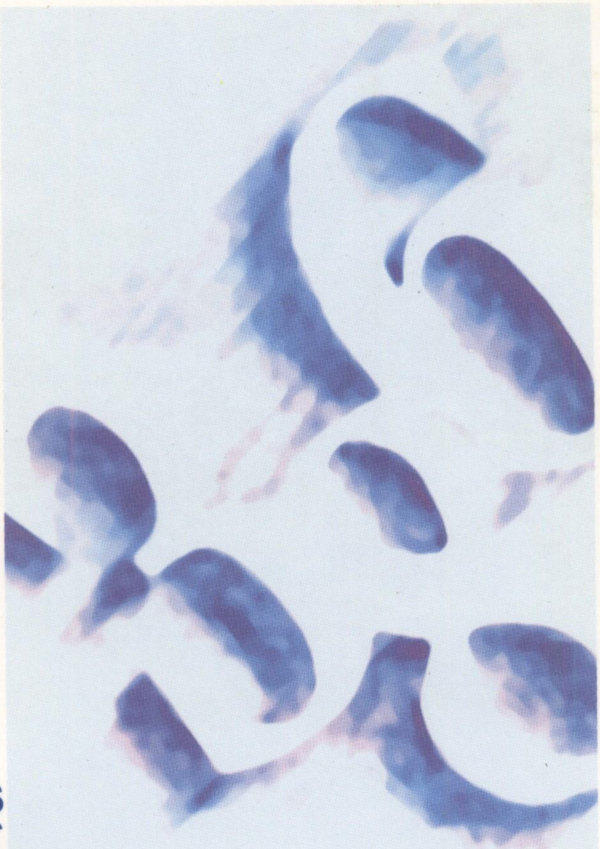


MS-DOS
ユーザーズ

386パソコン活用ブック

ソフトバンク

ISBN4-89052-303-0 C0055 P2300E
定価 2,300円(本体2,233円)



MS-DOS
ユーザーズ

386パソコン活用ブック

SOFT BANK ソフトバンク

ISBN4-89052-303-0 C0055 P2300E

定価 2,300円(本体2,233円)



本書の目次

ISBN4-89052-303-0 C0055 P2300E